Android Developer Fundamentals (Version 2) course

Last updated Tue Sep 11 2018

This course was created by the Google Developer Training team.

- For details about the course, including links to all the concept chapters, apps, and slides, see Android Developer Fundamentals (Version 2).

  developer.android.com/courses/adf-v2

Note: This course uses the terms "codelab" and "practical" interchangeably.

We advise you to use the online version of this course rather than this static PDF to ensure you are using the latest content.

See developer.android.com/courses/adf-v2.

Unit 4: Saving user data

This PDF contains a one-time snapshot of the lessons in **Unit4: Saving user data.**

Lessons in this unit

Lesson 9: Preferences and settings

9.1 P: Shared preferences

9.2 P: App settings

Lesson 10: Storing data with Room

10.1A P: Room, LiveData, and ViewModel

10.1B P: Room, LiveData, and ViewModel

Lesson 9.1: Shared preferences

Introduction

Shared preferences allow you to store small amounts of primitive data as key/value pairs in a file on the device. To get a handle to a preference file, and to read, write, and manage preference data, use the `SharedPreferences` class. The Android framework manages the shared preferences file itself. The file is accessible to all the components of your app, but it is not accessible to other apps.

The data you save to shared preferences is different from the data in the saved activity state, which you learned about in an earlier chapter:

- Data in a saved activity instance state is retained across activity instances in the same user session.
- Shared preferences persist across user sessions. Shared preferences persist even if your app stops and restarts, or if the device reboots.

Use shared preferences only when you need to save a small amount data as simple key/value pairs. To manage larger amounts of persistent app data, use a storage method such as the Room library or an SQL database.

What you should already know

You should be familiar with:

- Creating, building, and running apps in Android Studio.
- Designing layouts with buttons and text views.
- Using styles and themes.
- Saving and restoring activity instance state.

What you'll learn

You will learn how to:

- Identify what shared preferences are.
- Create a shared preferences file for your app.
- Save data to shared preferences, and read those preferences back again.

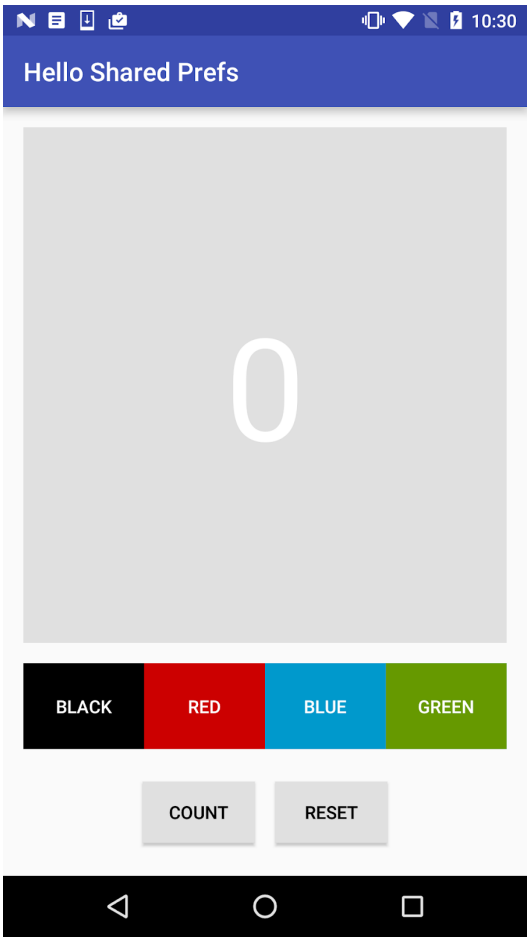
- Clear the data in the shared preferences.

What you'll do

- Update an app so it can save, retrieve, and reset shared preferences.

App overview

The HelloSharedPrefs app is another variation of the HelloToast app you created in Lesson 1. It includes buttons to increment the number, to change the background color, and to reset both the number and color to their defaults. The app also uses themes and styles to define the buttons.



You start with the starter app and add shared preferences to the main activity code. You also add a reset button that sets both the count and the background color to the default, and clears the preferences file.

Task 1: Explore HelloSharedPrefs

The complete starter app project for this practical is available at HelloSharedPrefs-Starter. In this task you load the project into Android Studio and explore some of the app's key features.

1.1 Open and run the HelloSharedPrefs project

1. Download the HelloSharedPrefs-Starter app and unzip the file.
2. Open the project in Android Studio, and build and run the app. Try these things:
    - Click the **Count** button to increment the number in the main text view.
    - Click any of the color buttons to change the background color of the main text view.
    - Rotate the device and note that both background color and count are preserved.
    - Click the **Reset** button to set the color and count back to the defaults.
3. Force-quit the app using one of these methods:
    - In Android Studio, select **Run > Stop 'app'** or click the Stop Icon  in the toolbar.
    - On the device, press the Recents button (the square button in the lower right corner). Swipe the HelloSharedPrefs app card to quit the app, or click the X in the right corner of the card. If you quit the app in this manner, wait a few seconds before starting it again so the system can clean up.
4. Re-run the app. The app restarts with the default appearance—the count is 0, and the background color is grey.

1.2 Explore the Activity code

1. Open `MainActivity`.
2. Examine the code and note these things:
    - The count (`mCount`) is defined as an integer. The `countUp()` onClick method increments this value and updates the main `TextView`.
    - The color (`mColor`) is also an integer that is initially defined as grey in the `colors.xml` resource file as `default_background`.
    - The `changeBackground()` onClick method gets the background color of the button that was clicked, and then sets the main text view to that color.
    - Both the `mCount` and `mColor` integers are saved to the instance state bundle in `onSaveInstanceState()`, and restored in `onCreate()`. The bundle keys for count and color are defined by private variables (`COUNT_KEY`) and (`COLOR_KEY`).

Task 2: Save and restore data to a shared preferences file

In this task you save the state of the app to a shared preferences file, and read that data back in when the app is restarted. Because the state data that you save to the shared preferences (the current count and color) are the **same** data that you preserve in the instance state, you don't have to do it twice. You can replace the instance state altogether with the shared preference state.

2.1 Initialize the preferences

1. Add member variables to the `MainActivity` class to hold the name of the shared preferences file, and a reference to a `SharedPreferences` object.

```
private SharedPreferences mPreferences;
private String sharedPrefFile =
   "com.example.android.hellosharedprefs";
```

You can name your shared preferences file anything you want to, but conventionally it has the same name as the package name of your app.

2. In the `onCreate()` method, initialize the shared preferences. Insert this code before the `if` statement:

```
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
```

The `getSharedPreferences()` method (from the activity `Context`) opens the file at the given filename (`sharedPrefFile`) with the mode `MODE_PRIVATE`.

Note: Older versions of Android had other modes that allowed you to create a world-readable or world-writable shared preferences file. These modes were deprecated in API 17, and are now **strongly discouraged** for security reasons. If you need to share data with other apps, consider using content URIs provided by a `FileProvider`.

Solution code for `MainActivity`, partial:

```
public class MainActivity extends AppCompatActivity {
   private int mCount = 0;
   private TextView mShowCount;
   private int mColor;

   private SharedPreferences mPreferences;
   private String sharedPrefFile =
      "com.example.android.hellosharedprefs";

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       mShowCount = (TextView) findViewById(R.id.textview);
       mColor = ContextCompat.getColor(this,
          R.color.default_background);
       mPreferences = getSharedPreferences(
          sharedPrefFile, MODE_PRIVATE);

       // …
    }
```

```
}
```

2.2 Save preferences in onPause()

Saving preferences is a lot like saving the instance state -- both operations set aside the data to a Bundle object as a key/value pair. For shared preferences, however, you save that data in the `onPause()` lifecycle callback, and you need a shared editor object (`SharedPreferences.Editor`) to write to the shared preferences object.

1. Add the `onPause()` lifecycle method to `MainActivity`.

```
@Override
protected void onPause(){
   super.onPause();

   // ...
}
```

2. In `onPause()`, get an editor for the `SharedPreferences` object:

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
```

A shared preferences editor is required to write to the shared preferences object. Add this line to `onPause()` after the call to `super.onPause()`.

3. Use the `putInt()` method to put both the `mCount` and `mColor` integers into the shared preferences with the appropriate keys:

```
preferencesEditor.putInt(COUNT_KEY, mCount);
preferencesEditor.putInt(COLOR_KEY, mColor);
```

The `SharedPreferences.Editor` class includes multiple "put" methods for different data types, including `putInt()` and `putString()`.

4. Call `apply()` to save the preferences:

```
preferencesEditor.apply();
```

The `apply()` method saves the preferences asynchronously, off of the UI thread. The shared preferences editor also has a `commit()` method to synchronously save the preferences. The `commit()` method is discouraged as it can block other operations.

1. Delete the entire `onSaveInstanceState()` method. Because the activity instance state contains the same data as the shared preferences, you can replace the instance state altogether.

Solution code for `MainActivity` `onPause()` method:

```
@Override
protected void onPause(){
    super.onPause();

    SharedPreferences.Editor preferencesEditor = mPreferences.edit();
    preferencesEditor.putInt(COUNT_KEY, mCount);
    preferencesEditor.putInt(COLOR_KEY, mColor);
    preferencesEditor.apply();
}
```

2.3 Restore preferences in onCreate()

As with the instance state, your app reads any saved shared preferences in the `onCreate()` method. Again, because the shared preferences contain the same data as the instance state, we can replace the state with the preferences here as well. Every time `onCreate()` is called -- when the app starts, on configuration changes -- the shared preferences are used to restore the state of the view.

1. Locate the part of the `onCreate()` method that tests if the `savedInstanceState` argument is null and restores the instance state:

```
if (savedInstanceState != null) {
   mCount = savedInstanceState.getInt(COUNT_KEY);
   if (mCount != 0) {
       mShowCountTextView.setText(String.format("%s", mCount));
   }
   mColor = savedInstanceState.getInt(COLOR_KEY);
   mShowCountTextView.setBackgroundColor(mColor);
}
```

2. Delete that entire block.
3. In the `onCreate()` method, in the same spot where the instance state code was, get the count from the preferences with the `COUNT_KEY` key and assign it to the `mCount` variable.

```
mCount = mPreferences.getInt(COUNT_KEY, 0);
```

When you read data from the preferences you don't need to get a shared preferences editor. Use any of the "get" methods on a shared preferences object (such as `getInt()` or `getString()` to retrieve preference data.

Note that the `getInt()` method takes two arguments: one for the key, and the other for the default value if the key cannot be found. In this case the default value is 0, which is the same as the initial value of `mCount`.

4. Update the value of the main `TextView` with the new count.

```
mShowCountTextView.setText(String.format("%s", mCount));
```

5. Get the color from the preferences with the `COLOR_KEY` key and assign it to the `mColor` variable.

```
    mColor = mPreferences.getInt(COLOR_KEY, mColor);
```

As before, the second argument to `getInt()` is the default value to use in case the key doesn't exist in the shared preferences. In this case you can just reuse the value of `mColor`, which was just initialized to the default background further up in the method.

6. Update the background color of the main text view.

```
mShowCountTextView.setBackgroundColor(mColor);
```

7. Run the app. Click the **Count** button and change the background color to update the instance state and the preferences.
8. Rotate the device or emulator to verify that the count and color are saved across configuration changes.
9. Force-quit the app using one of these methods:
    - In Android Studio, select **Run > Stop 'app.'**
    - On the device, press the Recents button (the square button in the lower right corner). Swipe the HelloSharedPrefs app card to quit the app, or click the X in the right corner of the card.
10. Re-run the app. The app restarts and loads the preferences, maintaining the state.

Solution code for `MainActivity` `onCreate()` method:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);

   // Initialize views, color, preferences
   mShowCountTextView = (TextView) findViewById(R.id.count_textview);
   mColor = ContextCompat.getColor(this, R.color.default_background);
   mPreferences = getSharedPreferences(
                        mSharedPrefFile, MODE_PRIVATE);

   // Restore preferences
   mCount = mPreferences.getInt(COUNT_KEY, 0);
   mShowCountTextView.setText(String.format("%s", mCount));
   mColor = mPreferences.getInt(COLOR_KEY, mColor);
   mShowCountTextView.setBackgroundColor(mColor);
}
```

2.4 Reset preferences in the reset() click handler

The reset button in the starter app resets both the count and color for the activity to their default values. Because the preferences hold the state of the activity, it's important to also clear the preferences at the same time.

1. In the `reset()` onClick method, after the color and count are reset, get an editor for the `SharedPreferences` object:

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
```

2. Delete all the shared preferences:

```
preferencesEditor.clear();
```

1. Apply the changes:

```
preferencesEditor.apply();
```

Solution code for the `reset()` method:

```
public void reset(View view) {
    // Reset count
    mCount = 0;
    mShowCountTextView.setText(String.format("%s", mCount));

    // Reset color
    mColor = ContextCompat.getColor(this, R.color.default_background);
    mShowCountTextView.setBackgroundColor(mColor);
```

```
    // Clear preferences
    SharedPreferences.Editor preferencesEditor = mPreferences.edit();
    preferencesEditor.clear();
    preferencesEditor.apply();

}
```

Solution code

Android Studio project: HelloSharedPrefs

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: Modify the HelloSharedPrefs app so that instead of automatically saving the state to the preferences file, add a second activity to change, reset, and save those preferences. Add a button to the app named Settings to launch that activity. Include toggle buttons and spinners to modify the preferences, and **Save** and **Reset** buttons for saving and clearing the preferences.

Summary

- The `SharedPreferences` class allows an app to store small amounts of primitive data as key-value pairs.
- Shared preferences persist across different user sessions of the same app.
- To write to the shared preferences, get a `SharedPreferences.Editor` object.
- Use the various "put" methods in a `SharedPreferences.Editor` object, such as `putInt()` or `putString()`, to put data into the shared preferences with a key and a value.

- Use the various "get" methods in a `SharedPreferences` object, such as `getInt()` or `getString()`, to get data out of the shared preferences with a key.
- Use the `clear()` method in a `SharedPreferences.Editor` object to remove all the data stored in the preferences.
- Use the `apply()` method in a `SharedPreferences.Editor` object to save the changes to the preferences file.

Related concept

The related concept documentation is in 9.0: Data storage and 9.1: Shared preferences.

Learn more

Android developer documentation:

- Data and file storage overview
- Save key-value data
- `SharedPreferences`
- `SharedPreferences.Editor`

Stack Overflow:

- How to use SharedPreferences in Android to store, fetch and edit values
- onSavedInstanceState vs. SharedPreferences

Homework

Build and run an app

Open the ScoreKeeper app that you created in the Android fundamentals 5.1: Drawables, styles, and themes lesson.

1. Replace the saved instance state with shared preferences for each of the scores.
2. To test the app, rotate the device to ensure that configuration changes read the saved preferences and update the UI.
3. Stop the app and restart it to ensure that the preferences are saved.
4. Add a **Reset** button that resets the score values to 0 and clears the shared preferences.

Answer these questions

Question 1

In which lifecycle method do you save the app state to shared preferences?

Question 2

In which lifecycle method do you restore the app state?

Question 3

Can you think of a case where it makes sense to have both shared preferences and instance state?

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- The app retains the scores when the device is rotated.
- The app retains the current scores after the app is stopped and restarted.
- The app saves the current scores to the shared preferences in the `onPause()` method.
- The app restores shared preferences in the `onCreate()` method.
- The app displays a **Reset** button that resets the scores to 0.

Make sure that the implementation of the on-click handler method for the **Reset** button does the following things:

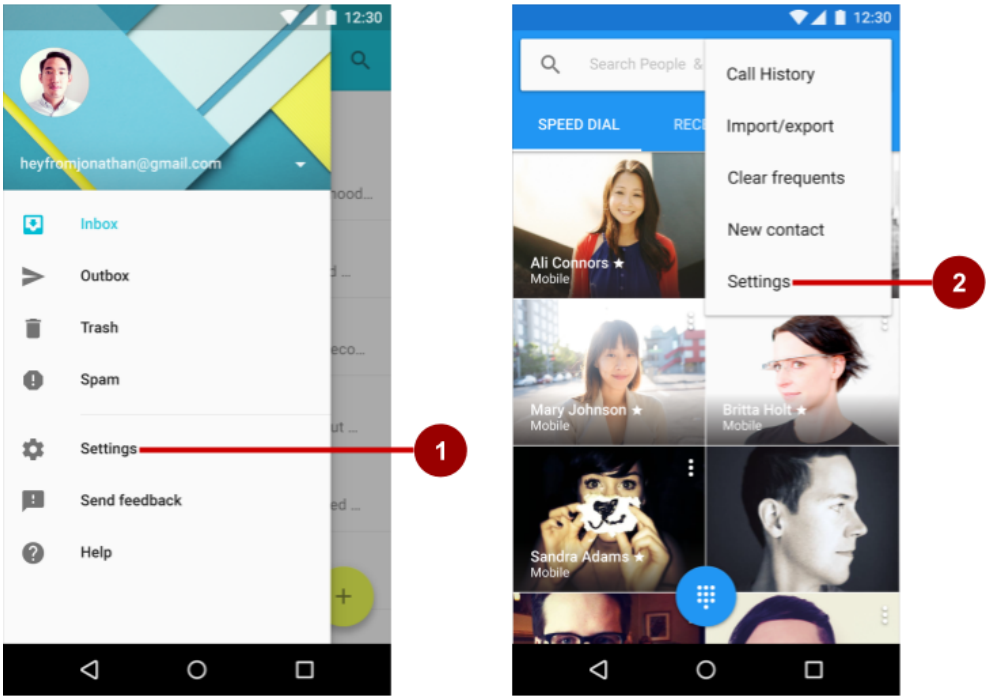
- Resets both score variables to 0.
- Updates both text views.
- Clears the shared preferences.

Lesson 9.2: App settings

Introduction

Apps often include settings that allow users to modify app features and behaviors. For example, some apps allow the user to set their home location, default units for measurements, and other settings that apply to the entire app. Users don't access settings frequently, because once a user changes a setting, such as a home location, they rarely need to go back and change it again.

Users expect to navigate to app settings by tapping **Settings** in side navigation, such as a navigation drawer as shown on the left side of the figure below, or in the options menu in the app bar, shown on the right side of the figure below.



In the figure above:

1. **Settings** in side navigation (a navigation drawer)
2. **Settings** in the options menu of the app bar

In this practical you add a settings activity to an app. Users will be able to navigate to the app settings by tapping **Settings**, which will be located in the options menu in the app bar.

What you should already know

You should be able to:

- Create an Android Studio project from a template and generate the main layout.
- Run apps on the emulator or a connected device.
- Create and edit UI elements using the layout editor and XML code.
- Extract string resources and edit string values.
- Access UI elements from your code using `findViewById()`.
- Handle a `Button` click.
- Display a `Toast` message.
- Add an `Activity` to an app.
- Create an options menu in the app bar.
- Add and edit the menu items in the options menu.
- Use styles and themes in a project.
- Use `SharedPreferences`.

What you'll learn

You will learn how to:

- Add a `Fragment` for managing settings.
- Create an XML resource file of settings with their attributes.
- Create navigation to the settings `Activity`.
- Set the default values of settings.
- Read the settings values changed by the user.
- Customize the Settings Activity template.

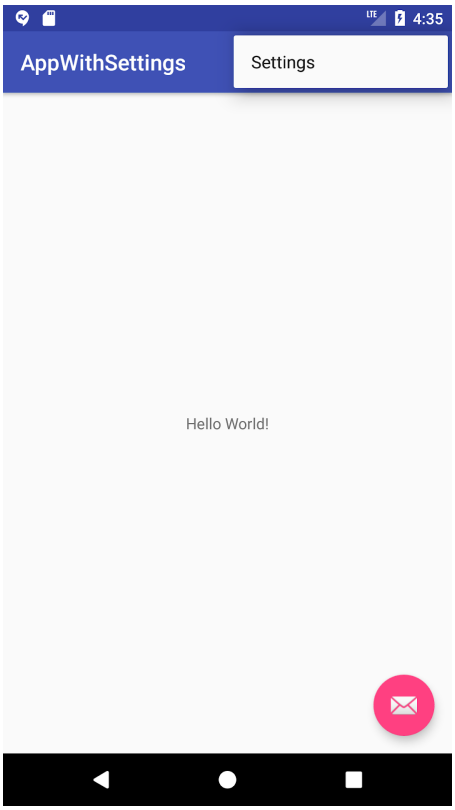
What you'll do

- Create an app that includes **Settings** in the options menu.
- Add a **Settings option** toggle switch.

- Add code to set the default value for the setting, and access the setting value after it has changed.
- Use and customize the Android Studio Settings Activity template.

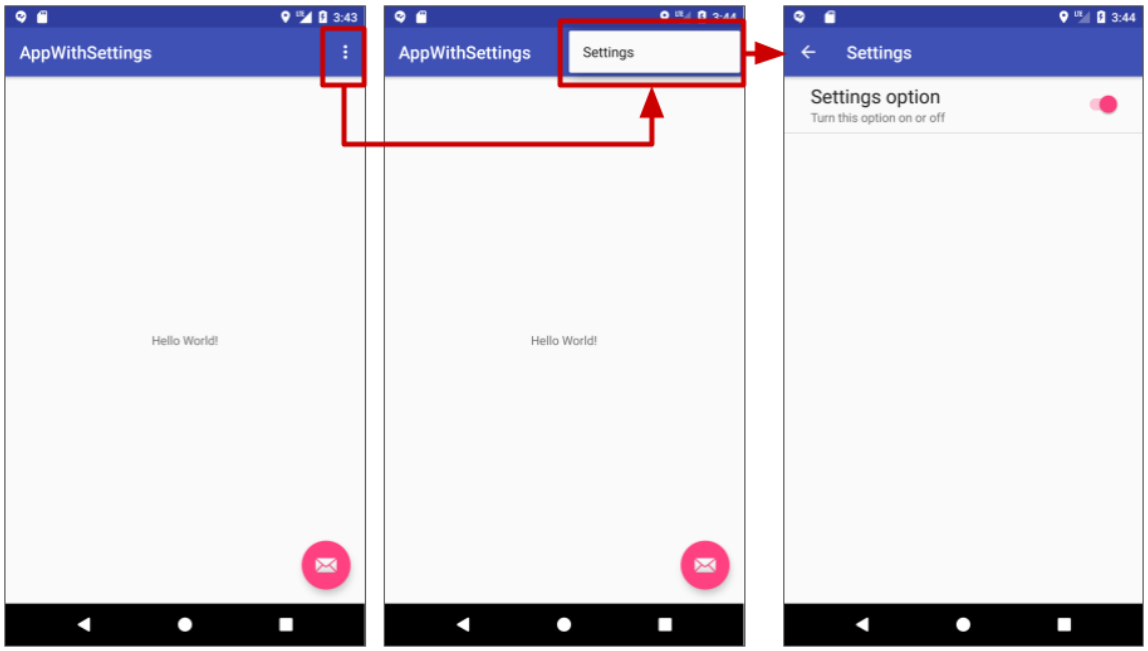
App overview

Android Studio provides a shortcut for setting up an options menu with **Settings**. If you start an Android Studio project for a phone or tablet using the Basic Activity template, the new app includes **Settings** as shown below:



The template also includes a floating action button in the lower right corner of the screen with an envelope icon. You can ignore this button for this practical, as you won't be using it.

You'll start by creating an app named AppWithSettings using the Basic Activity template, and you'll add a settings `Activity` that provides one toggle switch setting that the user can turn on or off:

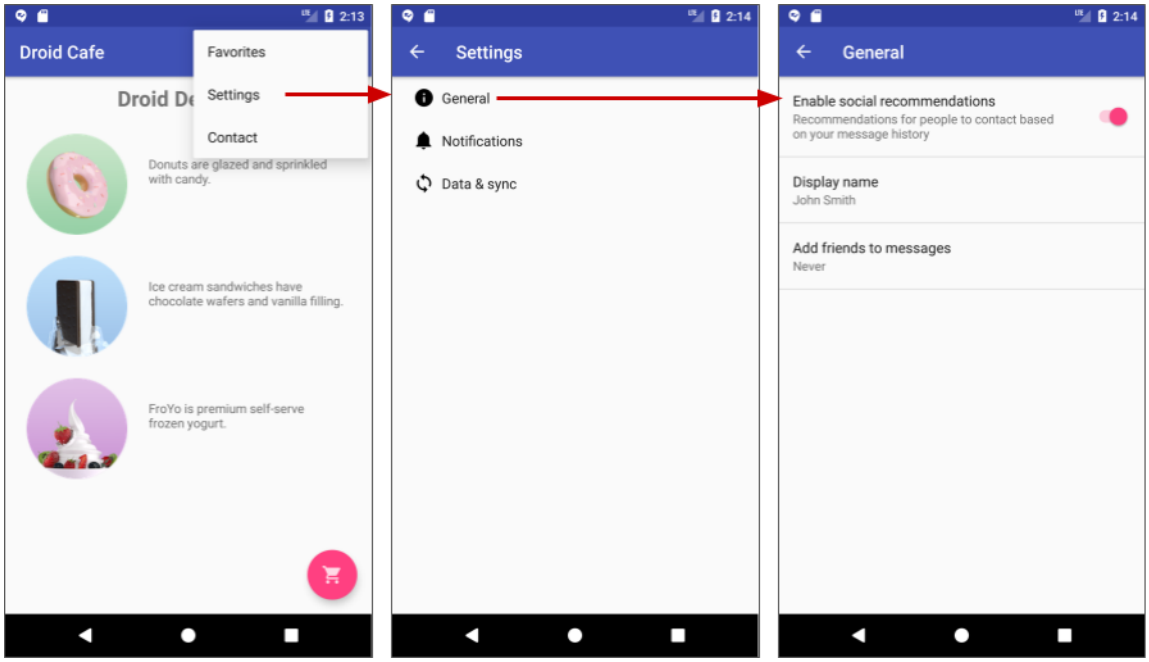


You will add code to read the setting and perform an action based on its value. For the sake of simplicity, the action will be to display a `Toast` message with the value of the setting.

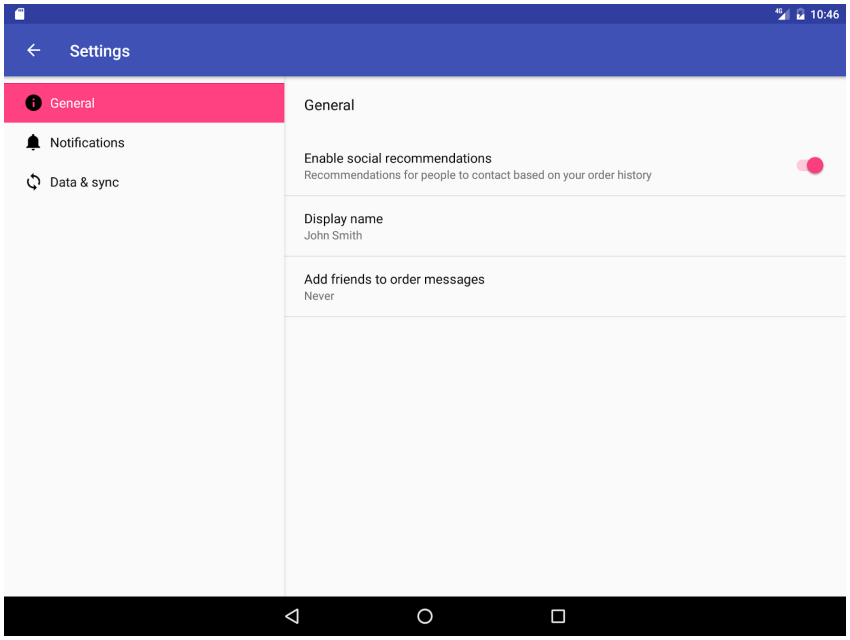
In the second task, you will add the standard Settings Activity template provided by Android Studio to the DroidCafeOptionsUp app you created in a previous lesson.

The Settings Activity template is pre-populated with settings you can customize for an app, and provides a different layout for phones and tablets:

- *Phones*: A main Settings screen with a header link for each group of settings, such as **General** for general settings, as shown below.



- *Tablets*: A master/detail screen layout with a header link for each group on the left (master) side, and the group of settings on the right (detail) side, as shown in the figure below.



To customize the template, you'll change the headers, setting titles, setting descriptions, and values for the settings.

The DroidCafeOptionsUp app was created in a previous lesson from the Basic Activity template, which provides an options menu in the app bar for placing the **Settings** option. You will customize the supplied Settings Activity template by changing a single setting's title, description, values, and default values. You will add code to read the setting's value after the user changes it, and display that value.

Task 1: Add a switch setting to an app

In this task, you do the following:

- Create a new project based on the Basic Activity template, which provides an options menu.
- Add a toggle switch (`SwitchPreference`) with attributes in a preference XML file.
- Add an activity for settings and a fragment for a specific setting. To maintain compatibility with `AppCompatActivity`, you use `PreferenceFragmentCompat` rather than `PreferenceFragment`. You also add the `android.support.v7.preference` `library`.
- Connect the **Settings** item in the options menu to the settings activity.

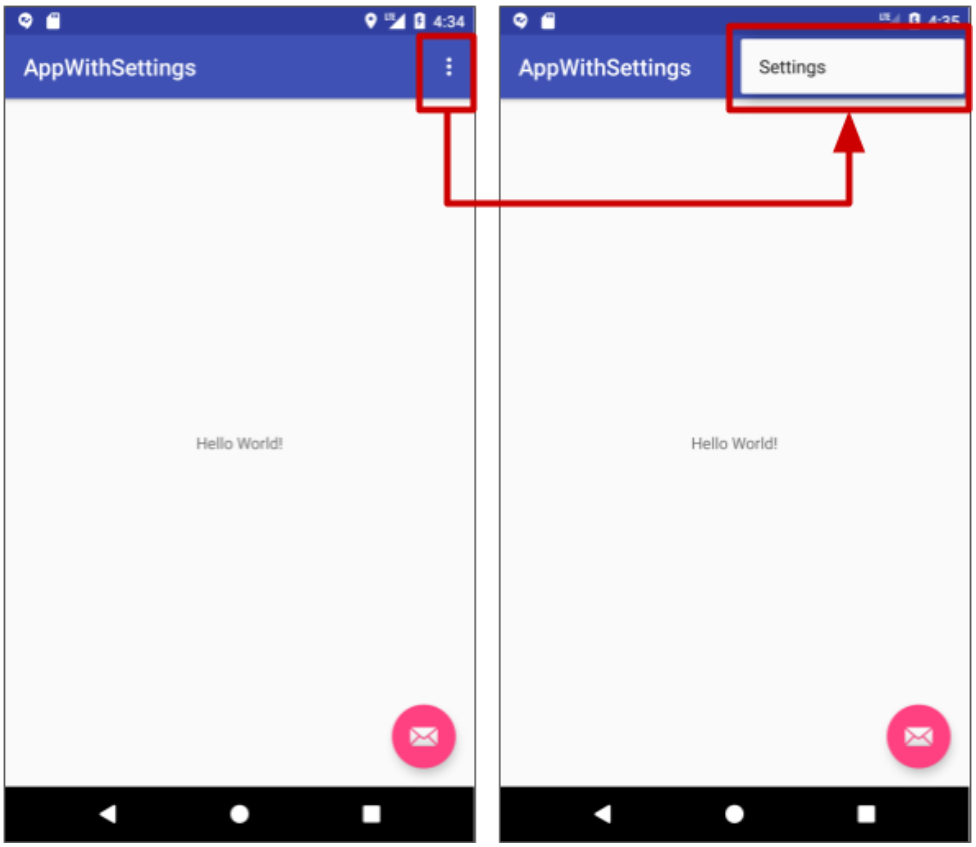
1.1 Create the project and add the xml directory and resource file

1. In Android Studio, create a new project with the following parameters:

Attribute	**Value**
Application Name	AppWithSettings
Company Name	android.example.com (or your own domain)
Project location	Path to your directory of projects
Phone and Tablet Minimum SDK	API15: Android 4.0.3 IceCreamSandwich
Template	Basic Activity
Activity Name	MainActivity

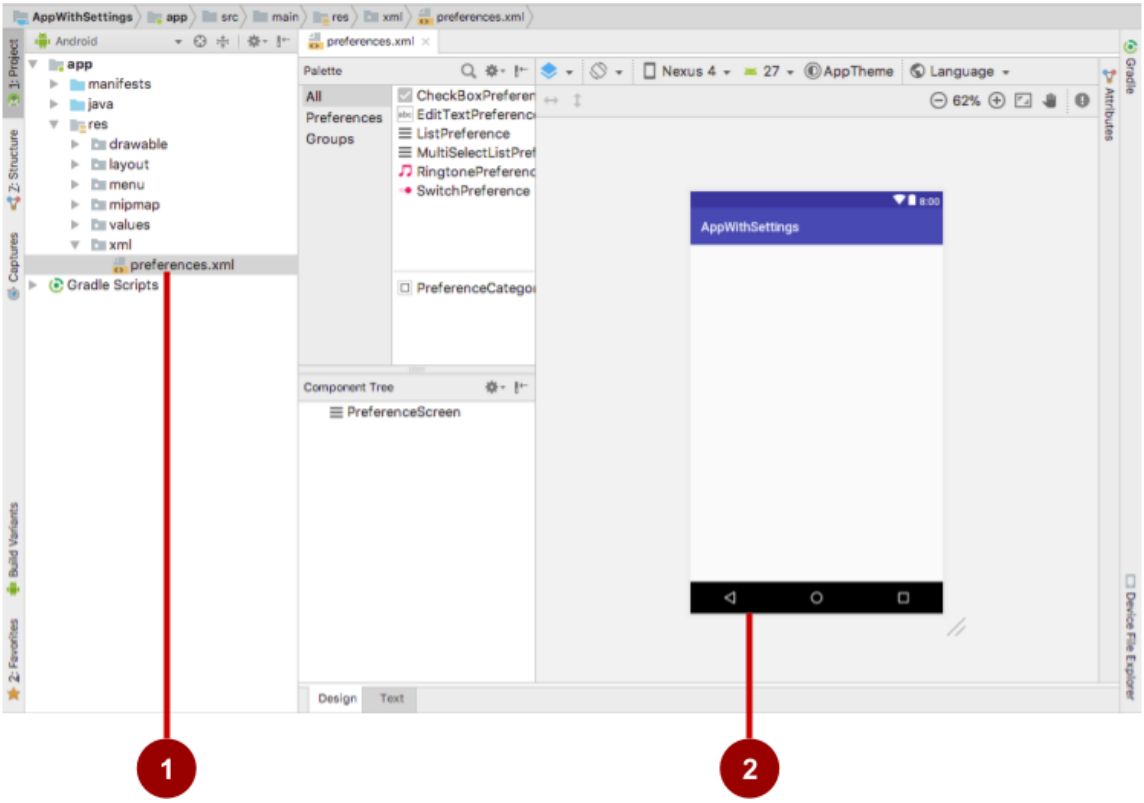
Layout Name	activity_main
Title	MainActivity

2. Run the app, and tap the overflow icon in the app bar to see the options menu, as shown in the figure below. The only item in the options menu is **Settings**.



3. You need to create a new resource directory to hold the XML file containing the settings. Select the **res** directory in the **Project > Android** pane, and choose **File > New > Android Resource Directory**. The New Resource Directory dialog appears.

4. In the Resource type drop-down menu, choose **xml**. The **Directory name** automatically changes to `xml`. Click **OK**.
5. The `xml` folder appears in the **Project > Android** pane inside the `res` folder. Select **xml** and choose **File > New > XML resource file** (or right-click **xml** and choose **New > XML resource file**).
6. Enter the name of the XML file, **preferences**, in the **File name** field, and click **OK**. The `preferences.xml` file appears inside the `xml` folder, and the layout editor appears, as shown in the figure below.

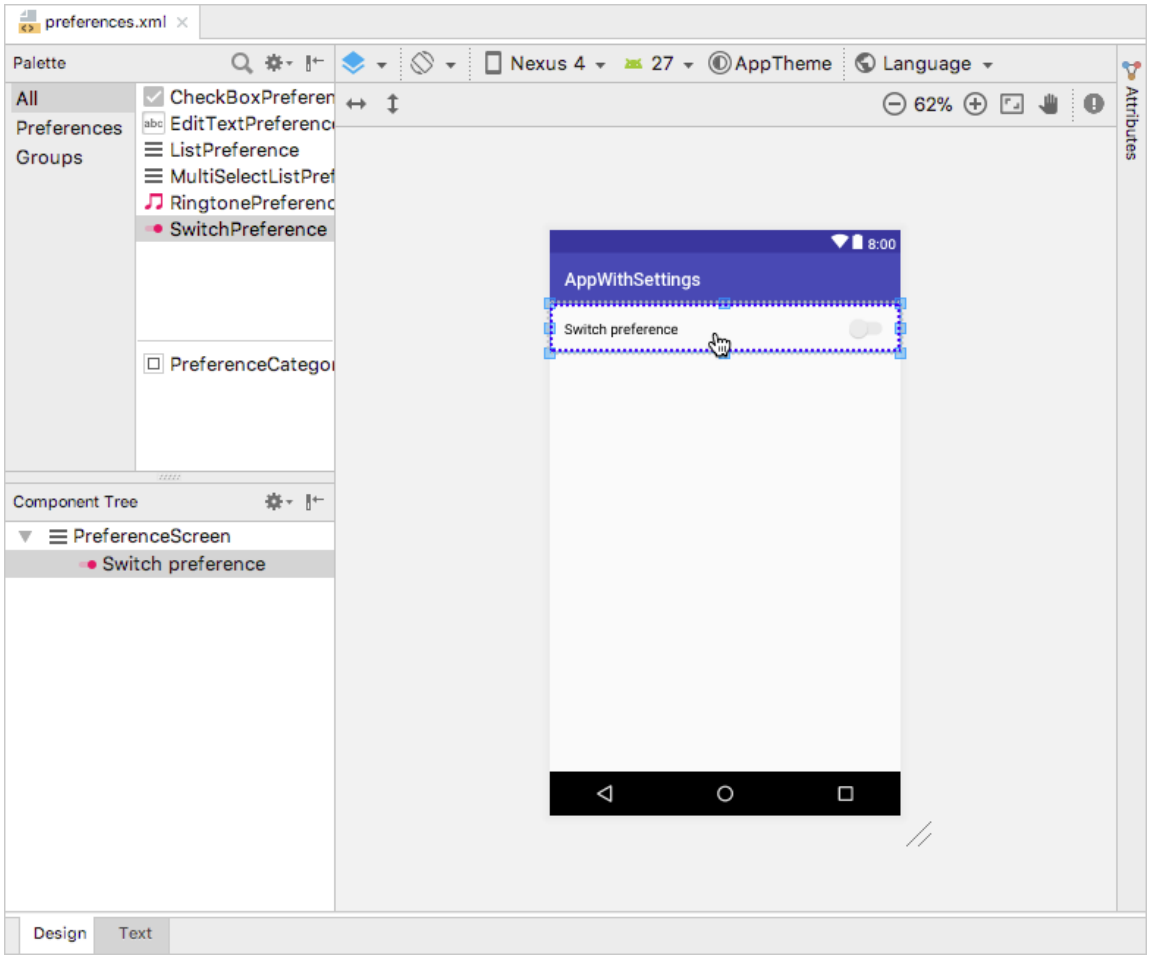


In the figure above:

1. The `preferences.xml` file inside the `xml` directory.
2. The layout editor showing the `preferences.xml` contents.

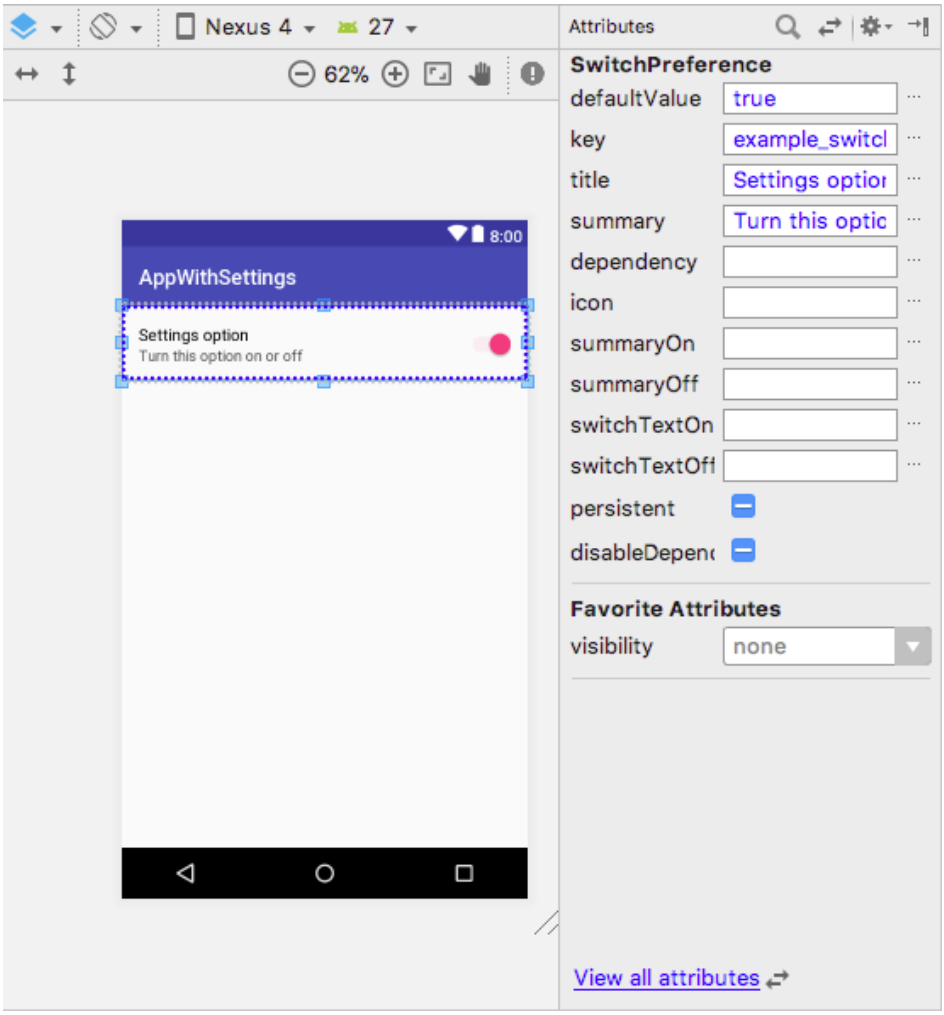
1.2 Add the XML preference and attributes for the setting

1. Drag a **SwitchPreference** from the **Palette** pane on the left side to the top of the layout, as shown in the figure below.



2. Change the values in the **Attributes** pane on the right side of the layout editor as follows, and as shown in the figure below:
    - defaultValue: **true**
    - key: **example_switch**

- title: **Settings option**
- summary: **Turn this option on or off**



3. Click the **Text** tab at the bottom of the layout editor to see the XML code:

```
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">

    <SwitchPreference
        android:defaultValue="true"
        android:key="example_switch"
```

```
        android:summary="Turn this option on or off"
        android:title="Settings option" />
</PreferenceScreen>
```

4. Extract the string resources for the `android:title` and `android:summary` attribute values to `@string/switch_title` and `@string/switch_summary`.

The XML attributes for a preferences are:

- `android:defaultValue`: The default value of the setting when the app starts for the first time.
- `android:title`: The title of the setting. For a `SwitchPreference`, the title appears to the left of the toggle switch.
- `android:key`: The key to use for storing the setting value. Each setting has a corresponding key-value pair that the system uses to save the setting in a default `SharedPreferences` file for your app's settings.
- `android:summary`: The text summary appears underneath the setting.

1.3 Use SwitchPreferenceCompat

In order to use the `PreferenceFragmentCompat` version of `PreferenceFragment`, you must also use the `android.support.v7` version of `SwitchPreference` (`SwitchPreferenceCompat`).

1. In the **Project > Android** pane, open the **build.gradle (Module: app)** file in the **Gradle Scripts** folder, and add the following to the `dependencies` section:

```
implementation 'com.android.support:preference-v7:26.1.0'
```

The statement shown above adds the `android.support.v7.preference` library in order to use the `PreferenceFragmentCompat` version of `PreferenceFragment`.

2. In the **preferences.xml** file in the **xml** folder, change `<SwitchPreference` in the code to `<android.support.v7.preference.SwitchPreferenceCompat`:

```
<android.support.v7.preference.SwitchPreferenceCompat
        android:defaultValue="true"
        android:key="example_switch"
        android:summary="@string/switch_summary"
        android:title="@string/switch_title" />
```

The `SwitchPreferenceCompat` line above may show a yellow light bulb icon with a warning, but you can ignore it for now.

3. Open the **styles.xml** file in the **values** folder, and add the following `preferenceTheme` item to the `AppTheme` declaration:

```
<item name="preferenceTheme">@style/PreferenceThemeOverlay</item>
```

In order to use `PreferenceFragmentCompat`, you must also declare `preferenceTheme` with the `PreferenceThemeOverlay` style to the app theme.

1.4 Add an Activity for settings

In order to create a settings `Activity` that provides a UI for settings, add an Empty Activity to the app. Follow these steps:

1. Select **app** at the top of the **Project > Android** pane, and choose **New > Activity > Empty Activity**.
2. Name the `Activity` **SettingsActivity**. Uncheck the **Generate Layout File** option (you don't need one), and leave unchecked the Launcher Activity option.
3. Leave the **Backwards Compatibility (AppCompat)** option checked. The Package name should already be set to **com.example.android.*projectname***.

4. Click **Finish**.

1.5 Add a Fragment for a specific setting

A `Fragment` is like a modular section of an `Activity`—it has its own lifecycle and receives its own input events, and you can add or remove a `Fragment` while the `Activity` is running. You use a specialized `Fragment` subclass to display a list of settings. The best practice is to use a regular `Activity` that hosts a `PreferenceFragment` that displays the app settings. `PreferenceFragment` provides a more flexible architecture for your app, compared to using an `Activity` for the preferences.

You will use `PreferenceFragmentCompat` rather than `PreferenceFragment` in order to maintain compatibility with `AppCompatActivity`.

In this step you will add a blank `Fragment` for a group of similar settings (*without* a layout, factory methods, or interface callbacks) to the app, and extend `PreferenceFragmentCompat`.

Follow these steps:

1. Select **app** again, and choose **New > Fragment > Fragment (Blank)**.
2. Name the fragment **SettingsFragment**. Uncheck the **Create layout XML?** option (you don't need one).
3. Uncheck the options to include fragment factory methods and interface callbacks.
4. The **Target Source Set** should be set to **main**.
5. Click **Finish**. The result is the following class definition in `SettingsFragment`:

```
public class SettingsFragment extends Fragment {

    public SettingsFragment() {
        // Required empty public constructor
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
             ViewGroup container, Bundle savedInstanceState) {
        TextView textView = new TextView(getActivity());
        textView.setText(R.string.hello_blank_fragment);
        return textView;
```

```
    }
}
```

6. Edit the class definition of `SettingsFragment` to extend `PreferenceFragmentCompat`:

```
public class SettingsFragment extends PreferenceFragmentCompat {
```

As you change the class definition so it matches the definition shown above, a red bulb appears in the left margin. Click the red bulb and choose **Implement methods**, and then choose **onCreatePreferences**. Android Studio creates the following `onCreatePreferences()` stub:

```
@Override
public void onCreatePreferences(Bundle
                        savedInstanceState, String rootKey) {
}
```

In order to extend the `Fragment`, Android Studio adds the following import statement:

```
import android.support.v7.preference.PreferenceFragmentCompat;
```

7. Delete the entire `onCreateView()` method in the fragment.

The reason why you are essentially replacing `onCreateView()` with `onCreatePreferences()` is because you will be adding this `SettingsFragment` to the existing `SettingsActivity` to display preferences, rather than showing a separate `Fragment` screen. Adding it to the existing `Activity` makes it easy to add or remove a `Fragment` while the `Activity` is running. The preference `Fragment` is rooted at the `PreferenceScreen` using `rootKey`.

You can safely delete the empty constructor from `SettingsFragment` as well, because the `Fragment` is not displayed by itself:

```
public SettingsFragment() {
        // Required empty public constructor
}
```

8. You need to associate with this `Fragment` the `preferences.xml` settings resource you created in a previous step. Add to the newly created `onCreatePreferences()` stub a call to `setPreferencesFromResource()` passing the id of the XML file (`R.xml.preferences`) and the `rootKey` to identify the preference root in `PreferenceScreen`:

```
setPreferencesFromResource(R.xml.preferences, rootKey);
```

The `onCreatePreferences()` method should now look like this:

```
@Override
public void onCreatePreferences(Bundle
                        savedInstanceState, String rootKey) {
    setPreferencesFromResource(R.xml.preferences, rootKey);
}
```

1.6 Display the Fragment in SettingsActivity

To display the `Fragment` in `SettingsActivity`, follow these steps:

1. Open **SettingsActivity**.
2. Add the following code to the end of the `onCreate()` method so that the `Fragment` is displayed as the main content:

```
getSupportFragmentManager().beginTransaction()
         .replace(android.R.id.content, new SettingsFragment())
         .commit();
```

The code above uses the typical pattern for adding a fragment to an activity so that the fragment appears as the main content of the activity:

- Use `getFragmentManager()` if the class extends `Activity` and the `Fragment` extends `PreferenceFragment`.
- Use `getSupportFragmentManager()` if the class extends `AppCompatActivity` and the `Fragment` extends `PreferenceFragmentCompat`.

The entire `onCreate()` method in `SettingsActivity` should now look like the following:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getSupportFragmentManager().beginTransaction()
                .replace(android.R.id.content, new SettingsFragment())
                .commit();
}
```

1.7 Connect the Settings menu item to SettingsActivity

Use an `Intent` to launch `SettingsActivity` from `MainActivity` when the user selects **Settings** from the options menu.

1. Open **MainActivity** and find the `if` block in the `onOptionsItemSelected()` method, which handles the tap on **Settings** in the options menu:

```
if (id == R.id.action_settings) {
   return true;
}
```

2. Add an `Intent` to the `if` block to launch `SettingsActivity`:

```
if (id == R.id.action_settings) {
    Intent intent = new Intent(this, SettingsActivity.class);
    startActivity(intent);
    return true;
}
```

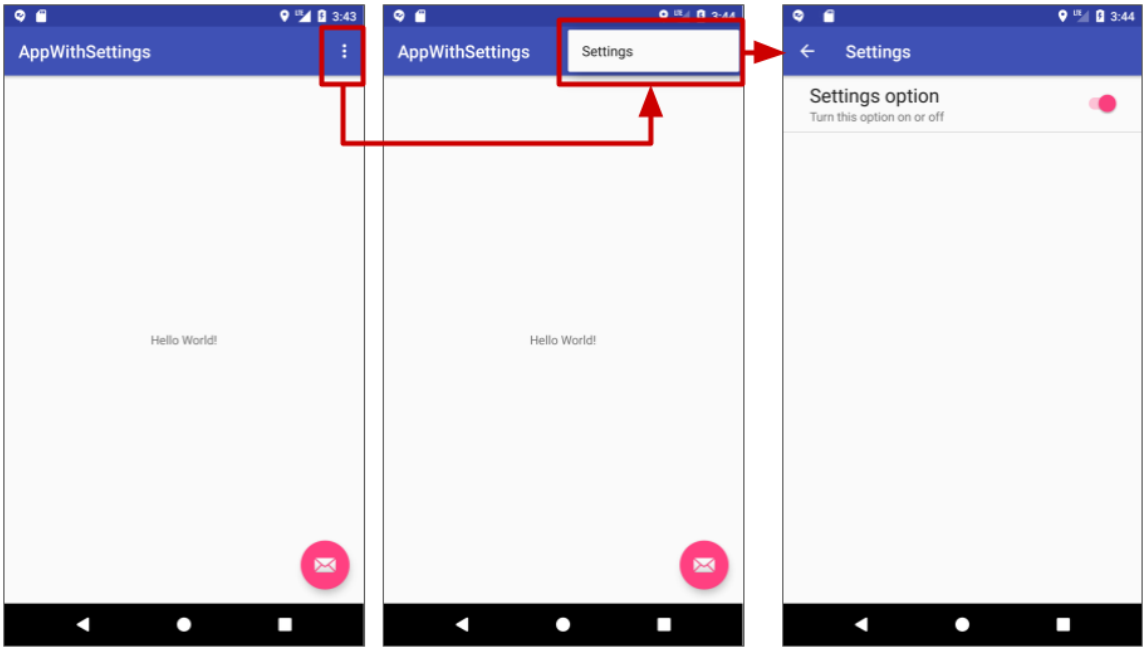
3. To add the app bar **Up** navigation button to `SettingsActivity`, you need to edit its declaration in the `AndroidManifest.xml` file to define the `SettingsActivity` `parent` as `MainActivity`. Open **AndroidManifest.xml** and find the `SettingsActivity` declaration:

```
<activity android:name=".SettingsActivity"></activity>
```

Change the declaration to the following:

```
<activity android:name=".SettingsActivity"
            android:label="Settings"
            android:parentActivityName=".MainActivity">
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value=".MainActivity"/>
</activity>
```

4. Run the app. Tap the overflow icon for the options menu, as shown on the left side of the figure below. Tap **Settings** to see the settings activity, as shown in the center of the figure below. Tap the **Up** button in the app bar of the settings activity, shown on the right side of the figure below, to return to the main activity.



1.8 Save the default values in shared preferences

Although the default value for the toggle switch setting has already been set in the `android:defaultValue` attribute (in Step 1.2 of this task), the app must save the default value in the `SharedPreferences` file for each setting when the user first opens the app. Follow these steps to set the default value for the toggle switch:

1. Open **MainActivity**.
2. Add the following to the end of the `onCreate()` method after the `FloatingActionButton` code:

```
android.support.v7.preference.PreferenceManager
            .setDefaultValues(this, R.xml.preferences, false);
```

The code above ensures that the settings are properly initialized with their default values. The `PreferenceManager.setDefaultValues()` method takes three arguments:

- The app `context`, such as `this`.
- The resource ID (`preferences`) for the XML resource file with one or more settings.
- A `boolean` indicating whether the default values should be set more than once. When `false`, the system sets the default values only if this method has never been called. As long as you set this third argument to `false`, you can safely call this method every time `MainActivity` starts without overriding the user's saved settings values. However, if you set it to `true`, the method will override any previous values with the defaults.

1.9 Read the changed settings value from shared preferences

When the app starts, the `MainActivity` `onCreate()` method can read the setting values that have changed, and use the changed values rather than the default values.

Each setting is identified using a key-value pair. The Android system uses this key-value pair when saving or retrieving settings from a `SharedPreferences` file for your app. When the user changes a setting, the system updates the corresponding value in the `SharedPreferences` file. To use the value of the setting, the app can use the key to get the setting from the `SharedPreferences` file using `PreferenceManager.getDefaultSharedPreferences()`.

Follow these steps to add that code:

1. Open **SettingsActivity** and create a `static` `String` variable to hold the key for the value:

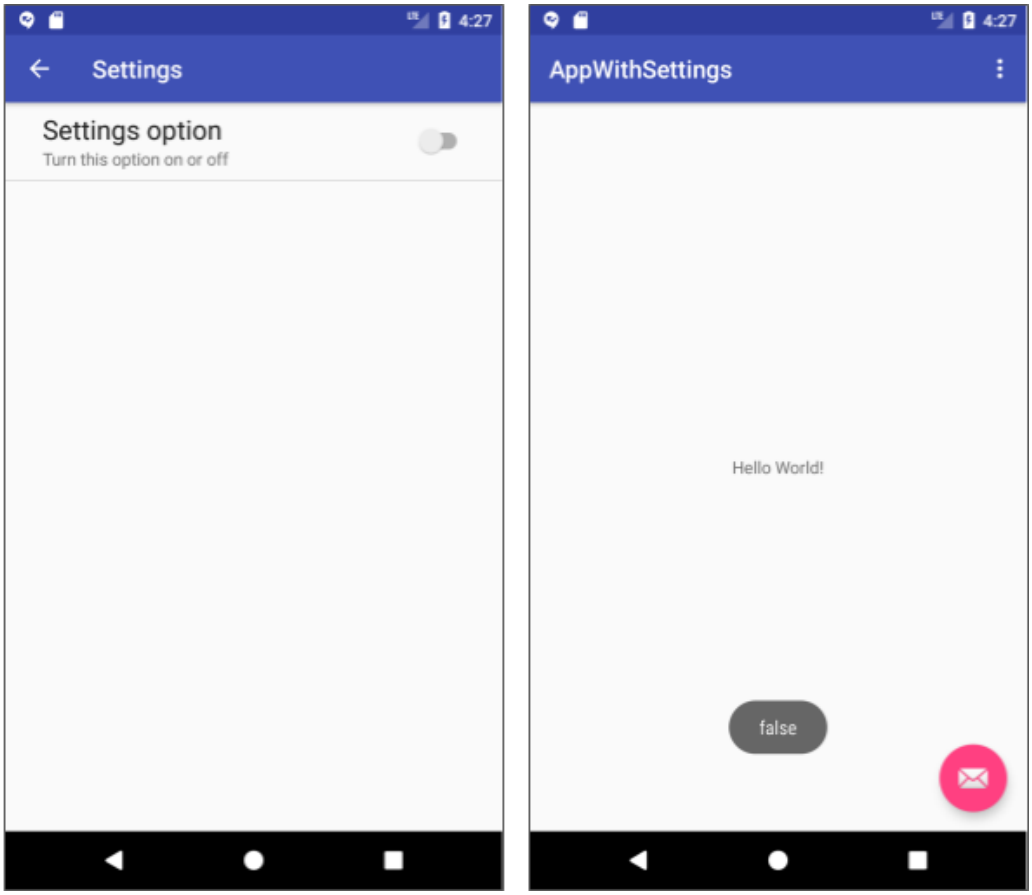
```
public static final String
            KEY_PREF_EXAMPLE_SWITCH = "example_switch";
```

2. Open **MainActivity** and add the following at end of the `onCreate()` method:

```
SharedPreferences sharedPref =
           android.support.v7.preference.PreferenceManager
                     .getDefaultSharedPreferences(this);
Boolean switchPref = sharedPref.getBoolean
           (SettingsActivity.KEY_PREF_EXAMPLE_SWITCH, false);
```

```
Toast.makeText(this, switchPref.toString(),
            Toast.LENGTH_SHORT).show();
```

3. Run the app and Tap **Settings** to see the Settings `Activity`.
4. Tap the setting to change the toggle from on to off, as shown on the left side of the figure below.
5. Tap the **Up** button in the Settings `Activity` to return to `MainActivity`. The `Toast` message should appear in `MainActivity` with the value of the setting, as shown on the right side of the figure below.
6. Repeat these steps to see the `Toast` message change as you change the setting.



The code snippet shown above uses the following:

- `android.support.v7.preference.PreferenceManager.getDefaultSharedPreferences(this)` to get the setting as a `SharedPreferences` object (`sharedPref`).
- `getBoolean()` to get the `Boolean` value of the setting that uses the key (`KEY_PREF_EXAMPLE_SWITCH` defined in `SettingsActivity`) and assign it to `switchPref`. If there is no value for the key, the `getBoolean()` method sets the setting value (`switchPref`) to `false`. For other values such as strings, integers, or floating point numbers, you can use the `getString()`, `getInt()`, or `getFloat()` methods respectively.

- `Toast.makeText()` and `show()` to display the value of the `switchPref` setting.

Whenever the `MainActivity` starts or restarts, the `onCreate()` method should read the setting values in order to use them in the app. The `Toast.makeText()` method would be replaced with a method that initializes the settings.

You now have a working Settings `Activity` in your app.

Task 1 solution code

Android Studio project: AppWithSettings

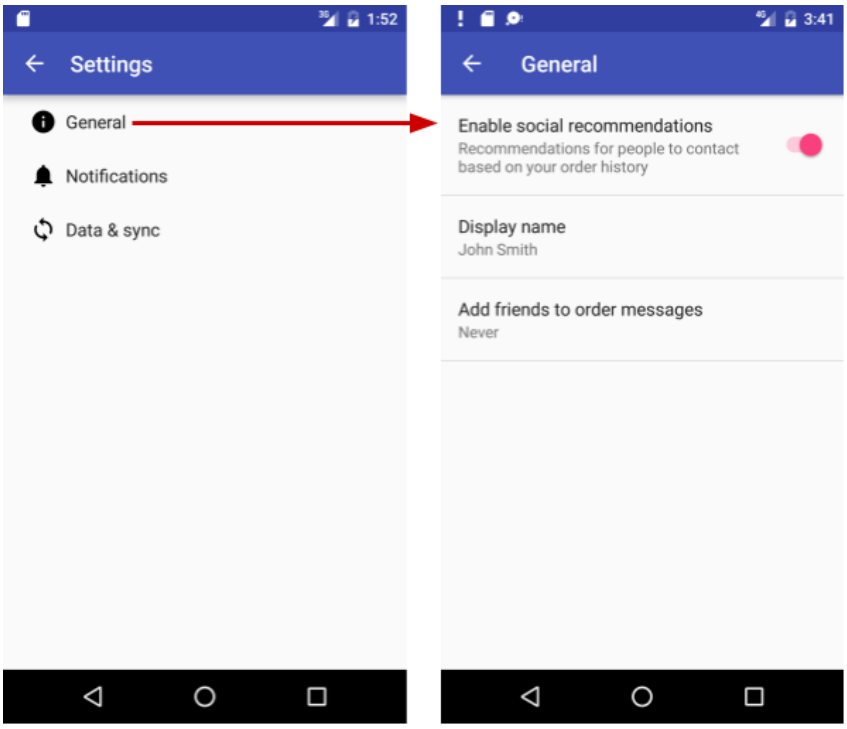
Task 2: Use the Settings Activity template

If you need to build several sub-screens of settings, and you want to take advantage of tablet-sized screens as well as maintain compatibility with older versions of Android for tablets, Android Studio provides a shortcut: the Settings Activity template.

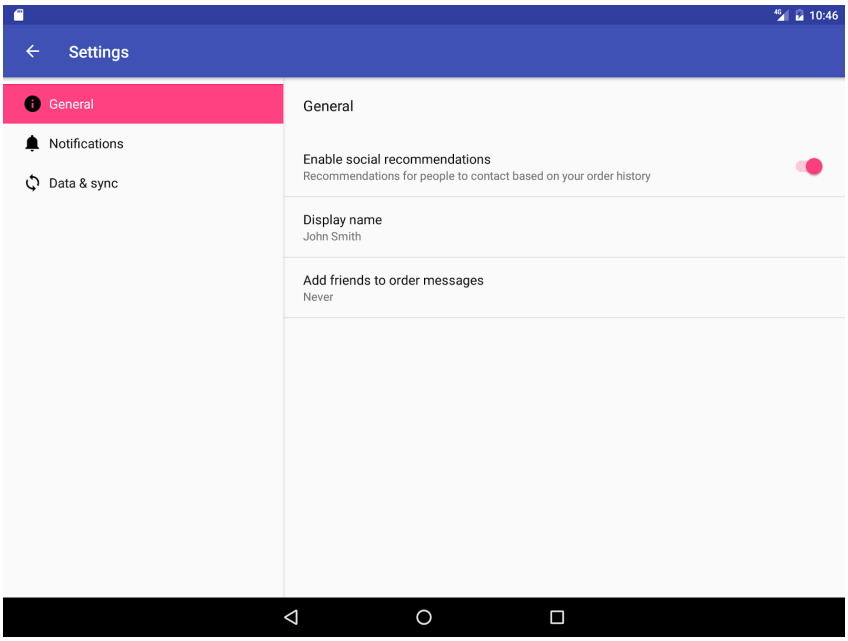
In the previous task you learned how to use an empty settings `Activity` and a blank `Fragment` in order to add a setting to an app. Task 2 will now show you how to use the Settings Activity template supplied with Android Studio to:

- Divide multiple settings into groups.
- Customize the settings and their values.

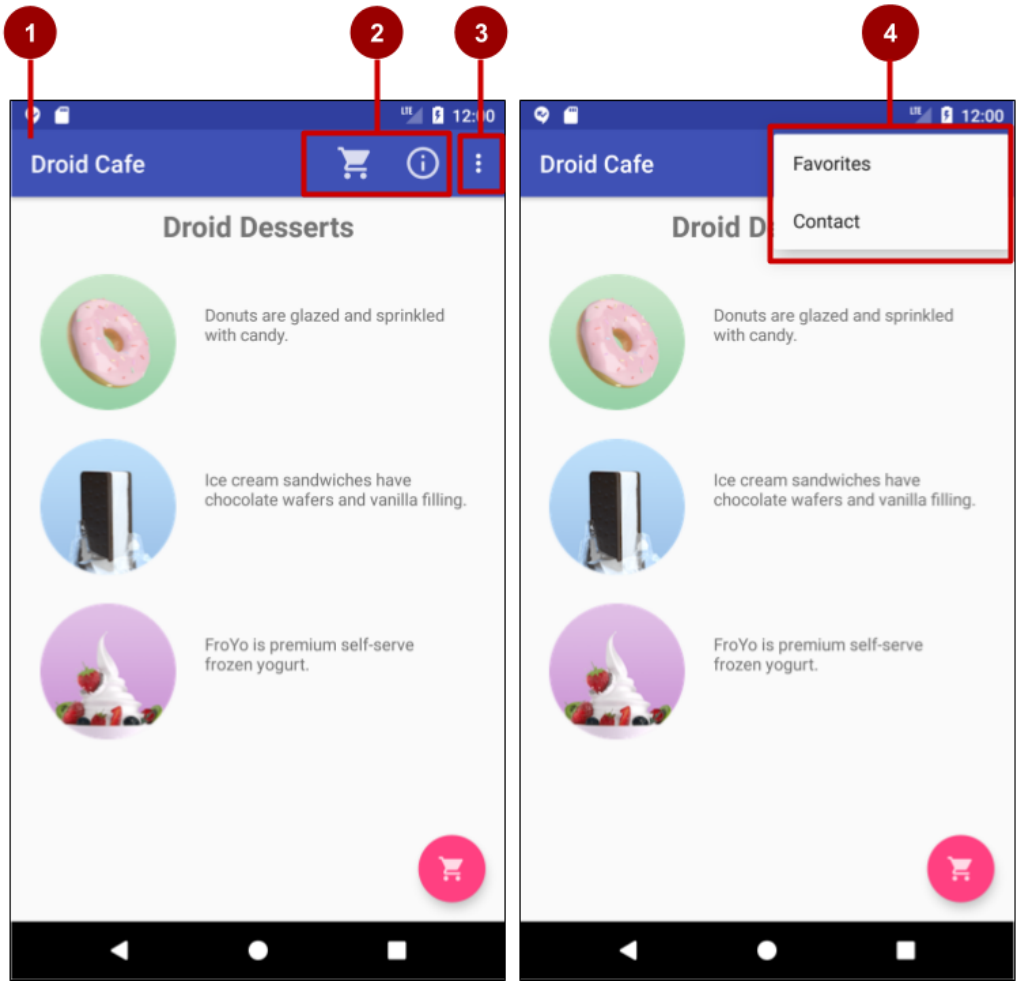
- Display a main Settings screen with a header link for each group of settings, such as General for general settings, as shown in the figure below.



- Display a master/detail screen layout with a header link for each group on the left (master) side, and the group of settings on the right (detail) side, as shown in the figure below.



In a previous practical you created an app called DroidCafeOptionsUp using the Basic Activity template, which provides an options menu in the app bar as shown below.



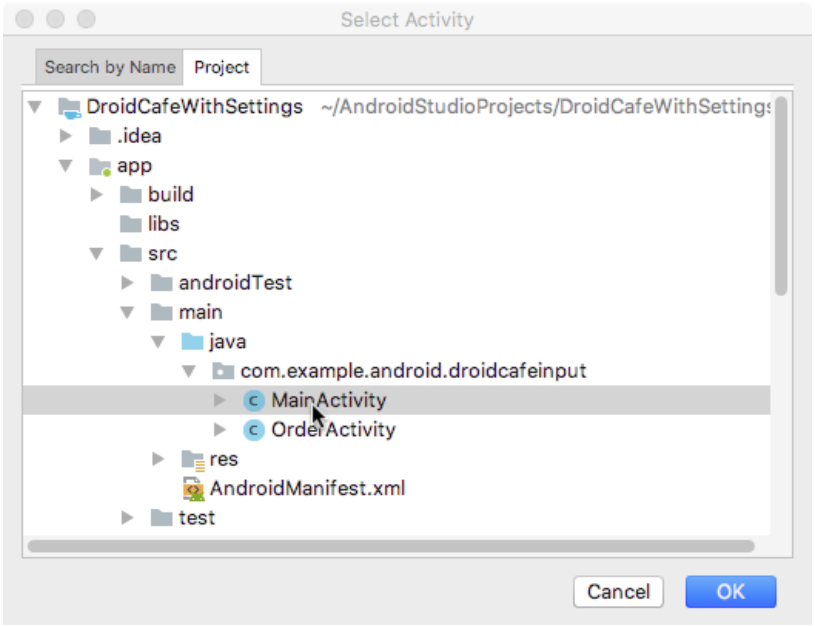
Legend for the figure above:

1. App bar
2. Options menu action icons
3. Overflow button
4. Options overflow menu

2.1 Explore the Settings Activity template

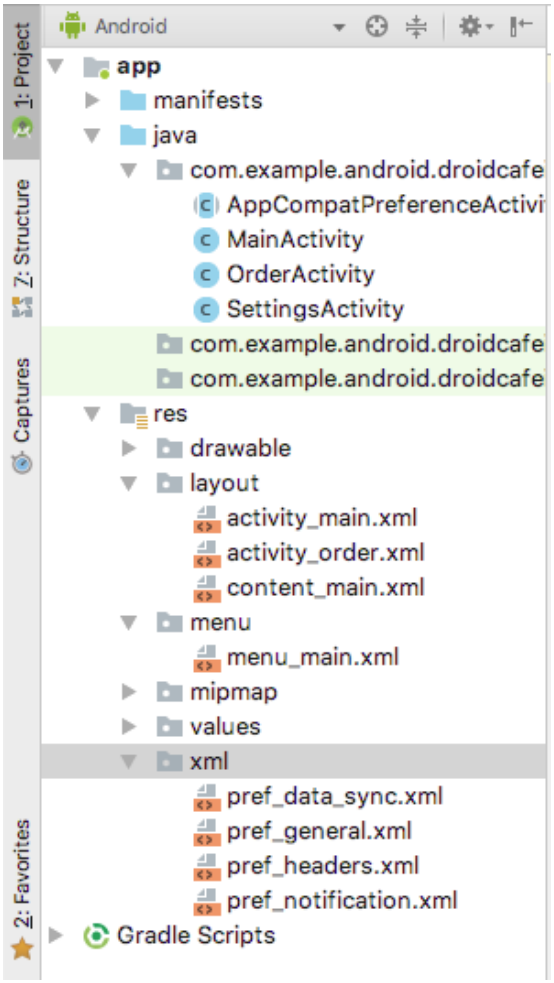
To include the Settings Activity template in an app project in Android Studio, follow these steps:

1. Copy the DroidCafeOptionsUp project folder, and rename it to **DroidCafeWithSettings**. Run the app to make sure it runs properly.
2. Select **app** at the top of the **Project > Android** pane, and choose **New > Activity > Settings Activity**.
3. In the dialog that appears, accept the Activity Name (**SettingsActivity** is the suggested name) and the Title (**Settings**).
4. Click the three dots at the end of the **Hierarchical Parent** menu, and click the **Project** tab in the Select Activity dialog that appears (refer to the figure below).
5. Expand **DroidCafeWithSettings > app > src > main > java > com.example.android.droidcafeinput** and select **MainActivity** as the parent activity, as shown in the figure below. Click **OK**.



You choose `MainActivity` as the parent so that the **Up** app bar button in the Settings Activity returns the user to the `MainActivity`. Choosing the parent `Activity` automatically updates the `AndroidManifest.xml` file to support **Up** button navigation.

6. Click **Finish**.
7. In the **Project > Android** pane, expand the **app > res > xml** folder to see the XML files created by the Settings Activity template.



You can open and then add to or customize the XML files for the settings you want:

- `pref_data_sync.xml`: `PreferenceScreen` layout for "Data & sync" settings.
- `pref_general.xml`: `PreferenceScreen` layout for "General" settings.
- `pref_headers.xml`: Layout of headers for the Settings main screen.
- `pref_notification.xml`: `PreferenceScreen` layout for "Notifications" settings.

The above XML layouts use various subclasses of the `Preference` class rather than `View`, and direct subclasses provide containers for layouts involving multiple settings. For example, `PreferenceScreen` represents a top-level `Preference` that is the root of a `Preference` hierarchy. The above files use `PreferenceScreen` at the top of each screen of settings. Other `Preference` subclasses for settings provide the appropriate UI for users to change the setting. For example:

- `CheckBoxPreference`: A checkbox for a setting that is either enabled or disabled.
- `ListPreference`: A dialog with a list of radio buttons.
- `SwitchPreference`: A two-state option that can be toggled (such as on/off or true/false).
- `EditTextPreference`: A dialog with an `EditText`.
- `RingtonePreference`: A dialog with ringtones on the device.

Tip: You can edit the XML files to change the default settings and customize the setting text. All strings used in the settings activity, such as setting titles, string arrays for lists, and setting descriptions, are defined as string resources in a `strings.xml` file. They are marked by comments such as the following:

```
<!-- Strings related to Settings -->
<!-- Example General settings -->
```

The Settings Activity template also creates:

- `SettingsActivity` in the `java/com.example.android.`*`projectname`* folder, which you can use as-is. This is the activity that displays the settings. `SettingsActivity` extends `AppCompatPreferenceActivity` for maintaining compatibility with older versions of Android.
- `AppCompatPreferenceActivity` in the `java/com.example.android.`*`projectname`* folder, which you use as is. This `Activity` is a helper class that `SettingsActivity` uses to maintain backward compatibility with previous versions of Android.

2.2 Add the Settings menu item and connect it to the activity

As you learned in another practical, you can edit the `menu_main.xml` file for the options menu to add or remove menu items.

1. Expand the **res** folder in the **Project > Android** pane, and open **menu_main.xml** file. Click the **Text** tab to show the XML code.
2. Add another menu item called **Settings** with the new resource id `action_settings`:

```
<item
    android:id="@+id/action_settings"
    android:orderInCategory="50"
```

```
    android:title="Settings"
    app:showAsAction="never" />
```

You specify `"never"` for the `app:showAsAction` attribute so that **Settings** appears only in the overflow options menu and not in the app bar itself, because it should not be used often. You specify `"50"` for the `android:orderInCategory` attribute so that **Settings** appears below **Favorites** (set to `"30"`) but above **Contact** (set to `"100"`).

3. Extract the string resource for `"Settings"` in the `android:title` attribute to the resource name `settings`.

4. Open **MainActivity**, and find the `switch case` block in the `onOptionsItemSelected()` method which handles the tap on items in the options menu. Shown below is a snippet of that method showing the first `case` (for `action_order`):

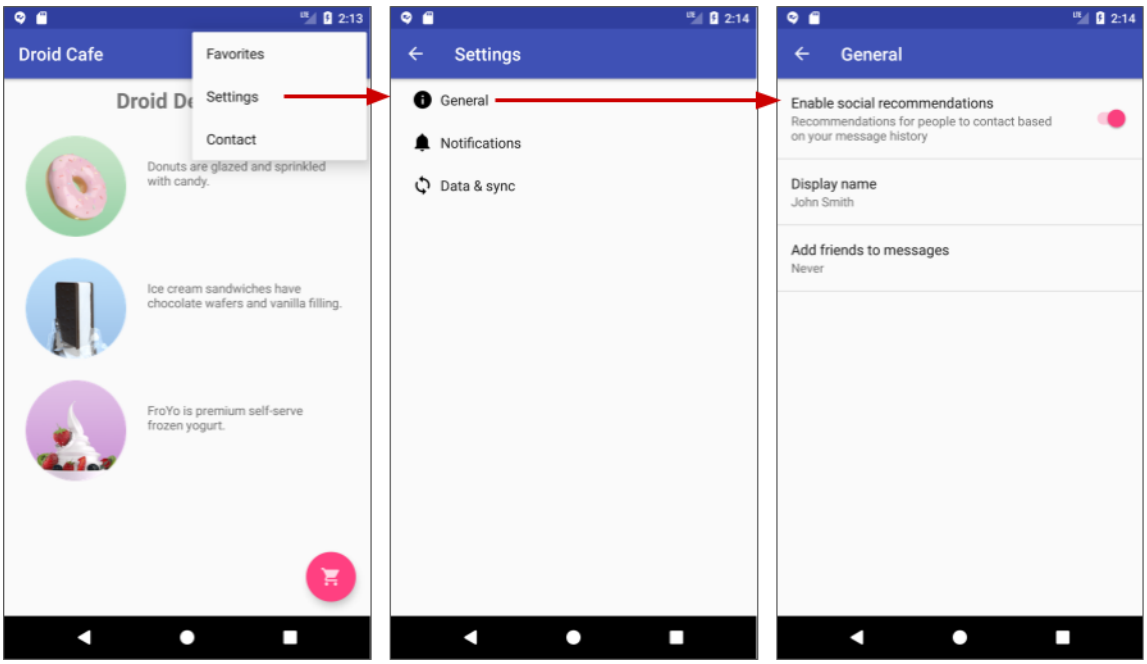
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_order:
            Intent intent = new Intent(MainActivity.this,
                            OrderActivity.class);
            intent.putExtra(EXTRA_MESSAGE, mOrderMessage);
            startActivity(intent);
            return true;
        case R.id.action_status:
            // Code for action_status and other cases...
        }

    return super.onOptionsItemSelected(item);
}
```

5. Note in the above code that the first `case` uses an `Intent` to launch `OrderActivity`. Add a new `case` for `action_settings` to the `switch case` block with similar `Intent` code to launch `SettingsActivity` (but without the `intent.putExtra`):

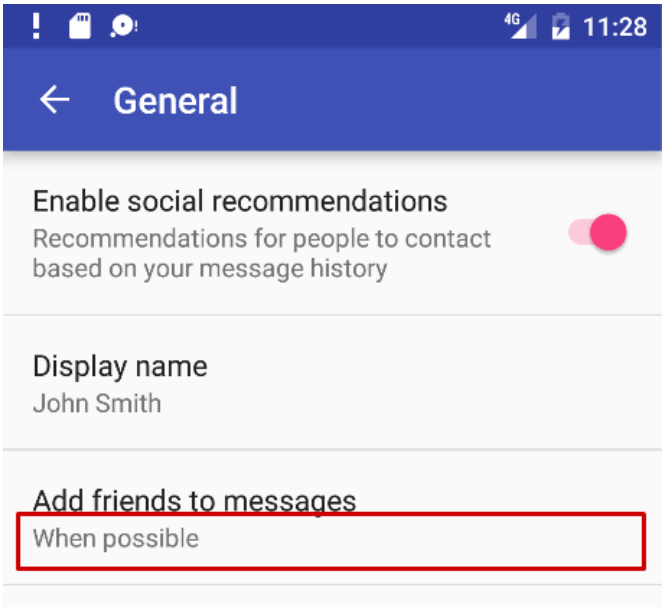
```
case R.id.action_settings:
    Intent settingsIntent = new Intent(this,
                            SettingsActivity.class);
```

```
    startActivity(settingsIntent);
    return true;
```

5. Run the app using a phone or emulator so that you can see how the Settings Activity template handles the phone screen size.
6. Tap the overflow icon for the options menu, and tap **Settings** to see the Settings `Activity`, as shown on the left side of the figure below.
7. Tap each setting header (**General**, **Notifications**, and **Data & sync**), as shown in the center of the figure below, to see the group of settings on each child screen of the Settings screen, shown on the right side of the figure below.
8. Tap the **Up** button in the Settings `Activity` to return to `MainActivity`.



You use the Settings Activity template code as-is. It not only provides layouts for phone-sized and tablet-sized screens, but also provides the function of listening to a settings change, and changing the summary to reflect the settings change. For example, if you change the "Add friends to messages" setting (the choices are **Always**, **When possible**, or **Never**), the choice you make appears in the summary underneath the setting:



In general, you need not change the Settings Activity template code in order to customize the `Activity` for the settings you want in your app. You can customize the settings titles, summaries, possible values, and default values without changing the template code, and even add more settings to the groups that are provided.

2.3 Customize the settings provided by the template

To customize the settings provided by the Settings Activity template, edit the string and string array resources in the `strings.xml` file and the layout attributes for each setting in the files in the `xml` directory.

In this step you will change the "Data & sync" settings.

1. Expand the **res > values** folder and open the **strings.xml** file. Scroll the contents to the `<!-- Example settings for Data & Sync -->` comment:

```
<!-- Example settings for Data & Sync -->
```

```
    <string name="pref_header_data_sync">Data &amp; sync</string>

    <string name="pref_title_sync_frequency">Sync frequency</string>
    <string-array name="pref_sync_frequency_titles">
        <item>15 minutes</item>
        <item>30 minutes</item>
        <item>1 hour</item>
        <item>3 hours</item>
        <item>6 hours</item>
        <item>Never</item>
    </string-array>
    <string-array name="pref_sync_frequency_values">
        <item>15</item>
        <item>30</item>
        <item>60</item>
        <item>180</item>
        <item>360</item>
        <item>-1</item>
    </string-array>

    <string-array name="list_preference_entries">
        <item>Entry 1</item>
        <item>Entry 2</item>
        <item>Entry 3</item>
    </string-array>

    <string-array name="list_preference_entry_values">
        <item>1</item>
        <item>2</item>
        <item>3</item>
    </string-array>

    <string-array name="multi_select_list_preference_default_value" />

    <string name="pref_title_system_sync_settings">System sync
settings</string>
```

2. Edit the `pref_header_data_sync` string resource, which is set to `Data &amp; sync` (the `&amp;` is HTML code for an ampersand). Change the value to **Account** (without quotation marks).

3. You should now refactor the resource name (the app will still work without refactoring the name, but refactoring makes the code easier to understand). **Right-click** (or **Control-click**) the **pref_header_data_sync** resource name choose **Refactor > Rename**. Change the name

to **pref_header_account**, click the option to search in comments and strings, and click **Refactor**.

4. You should also refactor the XML file name (the app will still work without refactoring the name, but refactoring makes the code easier to understand). **Right-click** (or **Control-click**) the **pref_data_sync** resource name in the **Project > Android** pane, and choose **Refactor > Rename**. Change the name to **pref_account**, click the option to search in comments and strings, and click **Refactor**.

5. Edit the `pref_title_sync_frequency` string resource (which is set to `Sync frequency`) to **Market**.

6. **Refactor > Rename** the `pref_title_sync_frequency` resource name to **pref_title_account** as you did previously.

7. **Refactor > Rename** the string array resource name `pref_sync_frequency_titles` to **pref_market_titles**.

8. Change each value in the `pref_market_titles` string array (`15 minutes`, `30 minutes`, `1 hour`, etc.) to be the titles of markets, such as **United States**, **Canada**, etc., rather than frequencies:

```
<string-array name="pref_market_titles">
        <item>United States</item>
        <item>Canada</item>
        <item>United Kingdom</item>
        <item>India</item>
        <item>Japan</item>
        <item>Other</item>
</string-array>
```

9. **Refactor > Rename** the string array resource name `pref_sync_frequency_values` to **pref_market_values**.

10. Change each value in the `pref_market_values` string array (`15`, `30`, `60`, etc.) to be values for the markets—abbreviations that correspond to the countries above, such as **US**, **CA**, etc.:

```
<string-array name="pref_market_values">
        <item>US</item>
        <item>CA</item>
        <item>UK</item>
```

```
        <item>IN</item>
        <item>JA0</item>
        <item>-1</item>
</string-array>
```

11. Scroll down to the `pref_title_system_sync_settings` string resource, and edit the resource (which is set to `System sync settings`) to **Account settings**.
12. **Refactor > Rename** the string array resource name `pref_title_system_sync_settings` to **pref_title_account_settings**.
13. Open the **pref_account.xml** file. The `ListPreference` in this layout defines the setting you just changed. Note that the string resources for the `android:entries`, `android:entryValues` and `android:title` attributes are now changed to the values you supplied in the previous steps:

```
<ListPreference
        android:defaultValue="180"
        android:entries="@array/pref_market_titles"
        android:entryValues="@array/pref_market_values"
        android:key="sync_frequency"
        android:negativeButtonText="@null"
        android:positiveButtonText="@null"
        android:title="@string/pref_title_account" />
```

14. Change the `android:defaultValue` attribute to **"US"**:

```
android:defaultValue="US"
```

Because the key for this setting preference (`"sync_frequency"`) is hard-coded elsewhere in the Java code, don't change the `android:key` attribute. Instead, keep using `"sync_frequency"` as the key for

this setting in this example. If you were thoroughly customizing the settings for a real-world app, you would take the time to change the hard-coded keys throughout the code.

Note: Why not use a string resource for the key? Because string resources can be localized for different languages using multiple-language XML files, and the key string might be inadvertently translated along with the other strings, which would cause the app to crash.

2.4 Add code to set the default values for the settings

To add code to set the default values for the settings, follow these steps:

1. Open **MainActivity** and find the `onCreate()` method.
2. Add the following `PreferenceManager.setDefaultValues` statements at the end of the `onCreate()` method:

```
PreferenceManager.setDefaultValues(this,
                R.xml.pref_general, false);
PreferenceManager.setDefaultValues(this,
                R.xml.pref_notification, false);
PreferenceManager.setDefaultValues(this,
                R.xml.pref_account, false);
```

The default values are already specified in the XML file with the `android:defaultValue` attribute, but the above statements ensure that the `SharedPreferences` file is properly initialized with the default values. The `setDefaultValues()` method takes three arguments:

- The app `context`, such as `this`.
- The resource ID for the settings layout XML file which includes the default values set by the `android:defaultValue` attribute.
- A `boolean` indicating whether the default values should be set more than once. When `false`, the system sets the default values *only* when this method is called for the first time. As long as you set this third argument to `false`, you can safely call this method every time your

`Activity` starts without overriding the user's saved settings values by resetting them to the default values. However, if you set it to `true`, the method will override any previous values with the defaults.

2.5 Add code to read values for the settings

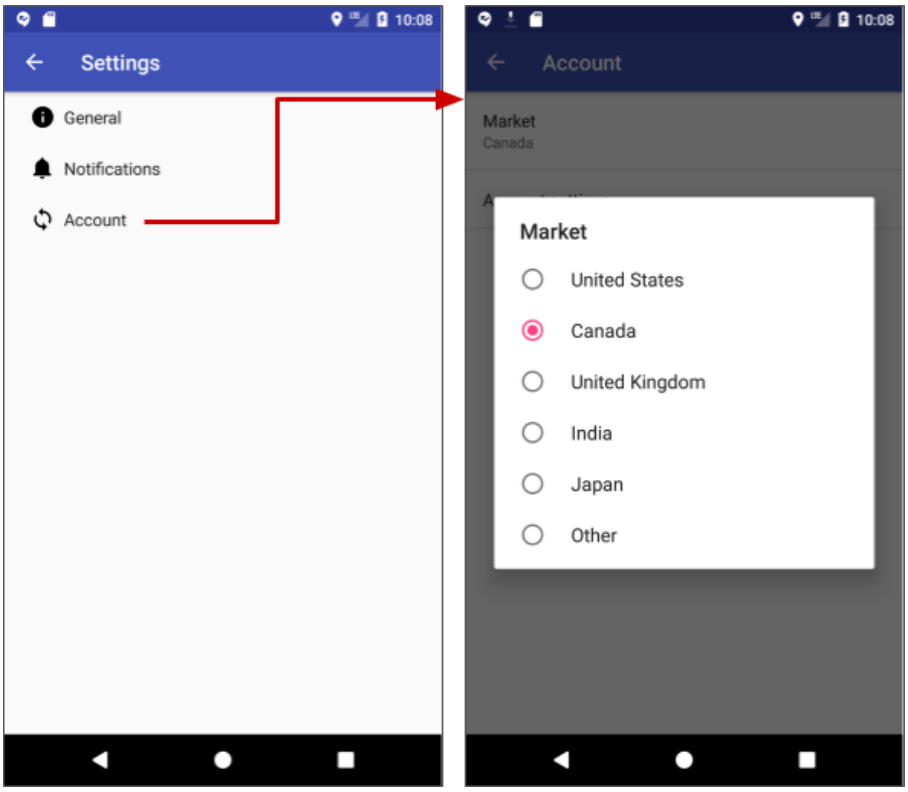
1. Add the following code at the end of the `MainActivity` `onCreate()` method. You can add it immediately after the code you added in the previous step to set the defaults for the settings:

```
SharedPreferences sharedPref = PreferenceManager
                        .getDefaultSharedPreferences(this);
String marketPref = sharedPref
                        .getString("sync_frequency", "-1");
displayToast(marketPref);
```

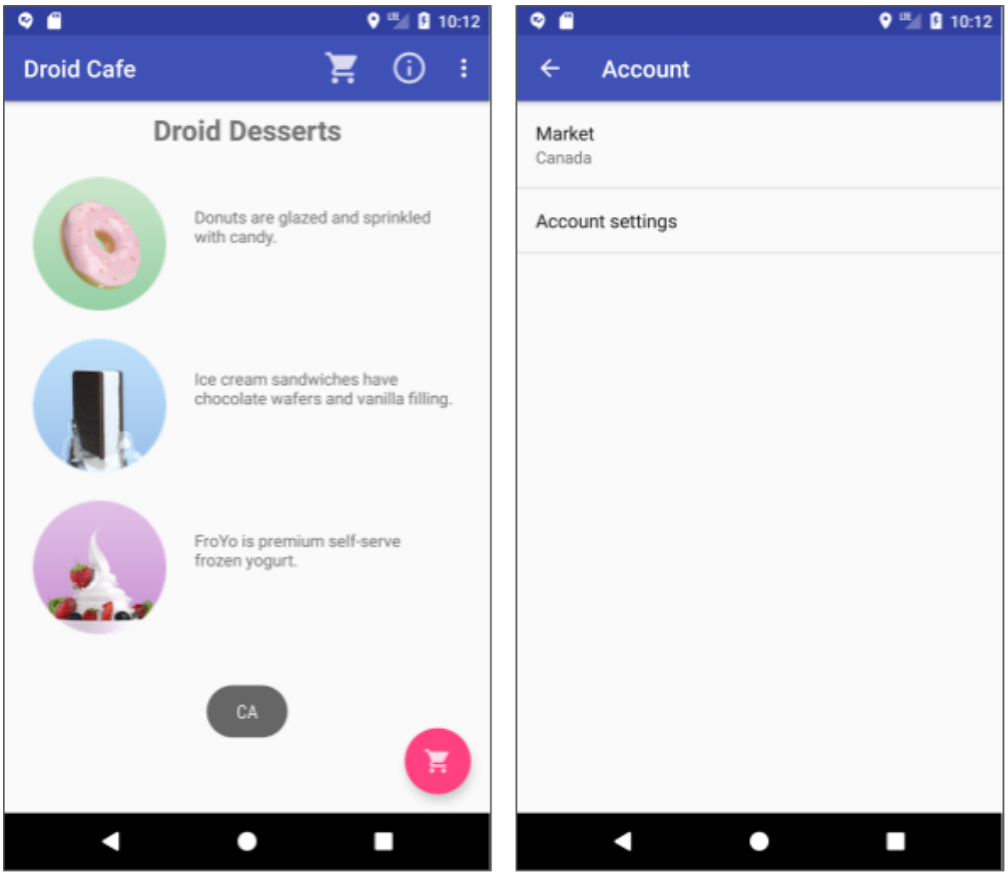
As you learned in the previous task, you use `PreferenceManager.getDefaultSharedPreferences(this)` to get the setting as a `SharedPreferences` object (`marketPref`). You then use `getString()` to get the string value of the setting that uses the key (`sync_frequency`), and assign it to `marketPref`. If there is no value for the key, the `getString()` method assigns the setting value of `marketPref` to `-1`, which is the value of `Other` in the `pref_market_values` array.

2. Run the app. When the app's main screen first appears, you see a `Toast` message at the bottom of the screen. The first time you run the app, you should see "-1" displayed in the `Toast` because you haven't changed the setting yet.

3. Tap **Settings** in the options menu, and tap **Account** in the Settings screen. Tap **Market**, and choose **Canada** as shown below:

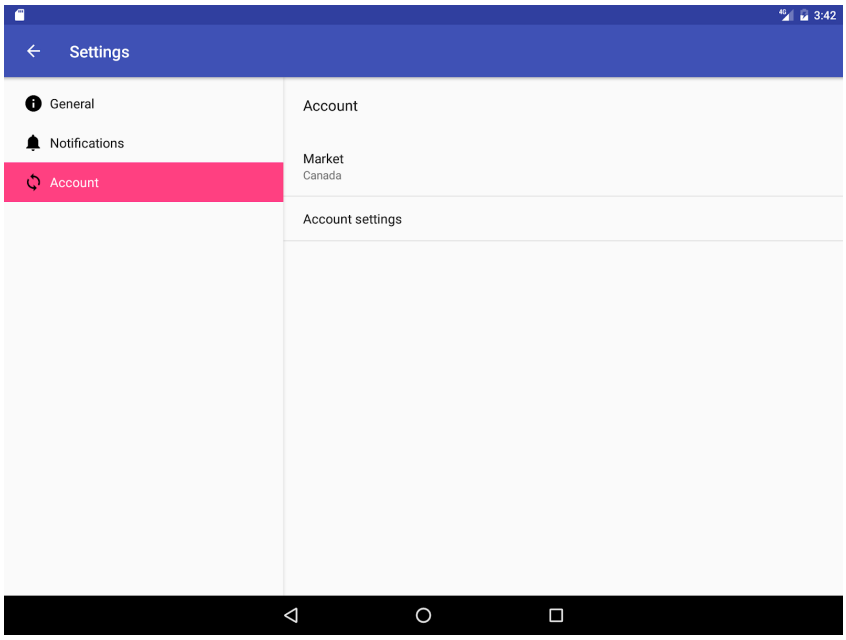


4. Tap the **Up** button in the app bar to return to the Settings screen, and tap it again to return to the main screen.
5. Run the app again from Android Studio. You should see a `Toast` message with "CA" (for Canada), and the **Market** setting is now set to Canada.



You have successfully integrated the Settings Activity with the app.

6. Now run the app on a tablet or tablet emulator. Because a tablet has a physically larger screen, the Android runtime takes advantage of the extra space. On a tablet, the settings and details are displayed on the same screen making it easier for users to manage their settings.



Task 2 solution code

Android Studio project: DroidCafeWithSettings

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: The DroidCafeWithSettings app displays the settings on a tablet-sized screen properly, but the **Up** button in the app bar doesn't return the user to the `MainActivity` as it does on a phone-sized screen. This is due to the three `onOptionsItemSelected()` methods—one for each `Fragment`—in `SettingsActivity`. It uses the following to restart the `SettingsActivity` when the user taps the **Up** button:

```
startActivity(new Intent(getActivity(), SettingsActivity.class));
```

The above is the appropriate action on phone screens in which Settings headers (**General**, **Notifications**, and **Account**) appear in a separate screen (`SettingsActivity`). After changing a setting, you want the user's tap on the **Up** button to take the user back to the Settings headers in `SettingsActivity`. A further tap on **Up** takes the user back to `MainActivity`.

However, on a tablet, the headers are always visible in the left pane (while the settings are in the right pane). As a result, tapping the **Up** button doesn't take the user to `MainActivity`.

Find a way to make the **Up** button work properly in `SettingsActivity` on tablet-sized screens.

Hint: While there are several ways to fix this problem, consider the following steps:

1. Add another `dimens.xml` file to specifically accommodate screen sizes larger than 600dp. When the app runs on a specific device, the appropriate `dimens.xml` file is chosen based on the qualifiers for the `dimens.xml` files. You can add another `dimens.xml` file with the **Smallest Screen Width** qualifier set to 600 dp (`sw600dp`), as you learned in the practical on supporting landscape and screen sizes, to specify any device with a large screen, such as a tablet.
2. Add the following `bool` resource between the `<resources>` and `</resources>` tags in the `dimens.xml (sw600dp)` file, which is automatically chosen for tablets:

```
<bool name="isTablet">true</bool>
```

3. Add the following `bool` resource to the standard `dimens.xml` file, which is chosen when the app runs on any device that is *not* large:

```
<bool name="isTablet">false</bool>
```

4. In `SettingsActivity`, you can add to all three `onOptionsItemSelected()` methods (one for each `Fragment`) an `if else` block that checks to see if `isTablet` is true. If it is, your code can redirect the **Up** button action to `MainActivity`.

Challenge solution code

Android Studio project: DroidCafeWithSettingsChallenge

Summary

Users expect to navigate to app settings by tapping **Settings** in side navigation, such as a navigation drawer, or in the options menu in the app bar.

To provide user settings for your app, provide an `Activity` for settings:

- Use an `Activity` that hosts a `PreferenceFragment` to display the app settings.
- To maintain compatibility with `AppCompatActivity` and the `android.support.v7.preference library`, use `PreferenceFragmentCompat` rather than `PreferenceFragment`.

Show each fragment in the settings activity:

- If the activity class extends `Activity` and the fragment class extends `PreferenceFragment`, use `getFragmentManager()`.
- If the activity class extends `AppCompatActivity` and the fragment class extends `PreferenceFragmentCompat`, use `getSupportFragmentManager()`.
- To associate the `preferences.xml` settings resource with the fragment, use `setPreferencesFromResource()`.
- To set the default values for settings, use `PreferenceManager.setDefaultValues()`.
- To connect the **Settings** menu item to the settings activity, use an `Intent`.

Add XML resource files for the settings:

1. Create a new resource directory (**File > New > Android Resource Directory**).
2. In the **Resource type** drop-down menu, select **xml**. The `xml` folder appears inside the `res` folder.
3. Click on **xml** and select **File > New > XML resource file**.

4. Enter **preferences** as the name of the XML file. The `preferences.xml` file appears inside the `xml` folder.

Add UI controls such as toggle switches, with attributes in a preferences XML file:

- To maintain compatibility with `AppCompatActivity`, use the `android.support.v7.preference library` version. For example, use `SwitchPreferenceCompat` for toggle switches.

Use attributes with each UI element for settings:

- `android:defaultValue` is the value of the setting when the app starts for the first time.
- `android:title` is the user-visible setting title.
- `android:key` is the key used for storing the setting value.
- `android:summary` is the user-visible text that appears under the setting.

Save and read settings values:

- When the app starts, the `MainActivity` `onCreate()` method can read the setting values that have changed, and use the changed values rather than the default values.
- Each setting is identified using a key-value pair. The Android system uses this key-value pair when saving or retrieving settings from a `SharedPreferences` file for your app. When the user changes a setting, the system updates the corresponding value in the `SharedPreferences` file.
- To use the value of the setting, your app can use the key to get the setting from the `SharedPreferences` file.
- Your app reads settings values from `SharedPreferences` using `PreferenceManager.getDefaultSharedPreferences()`, and obtains each setting value using `.getString`, `.getBoolean`, etc.

Related concepts

The related concept documentation is in 9.2: App settings.

Learn more

Android Studio documentation: Android Studio User Guide

Android developer documentation:

- Settings (overview)
- `Preference`
- `PreferenceFragment`
- `PreferenceFragmentCompat`
- `Fragment`
- `SharedPreferences`
- Save key-value data
- Support different screen sizes

Material Design specification: Android settings

Stack Overflow:

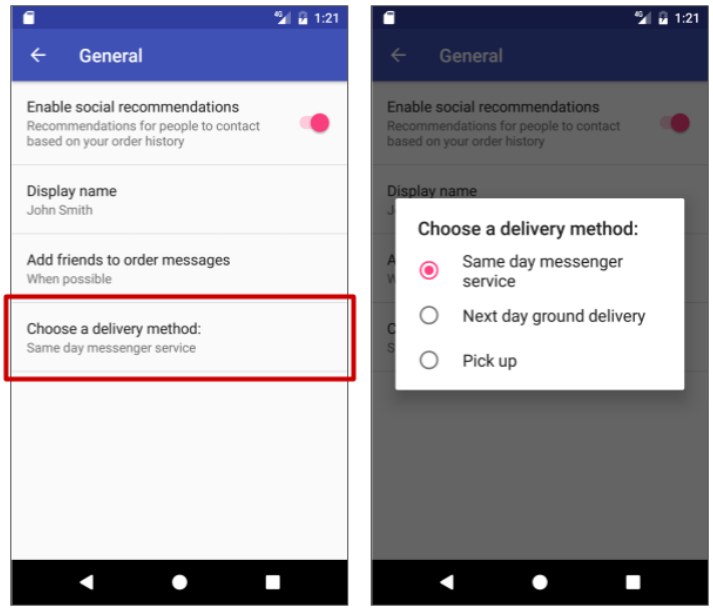
- How does one get dimens.xml into Android Studio?
- Determine if the device is a smartphone or tablet?

Homework

Build and run an app

Open the DroidCafeWithSettings app project.

1. Add a `ListPreference` (a dialog with radio buttons) to the general settings. Put the dialog in the **General** settings screen, below the "Add friends to order messages" `ListPreference`.
2. Edit the string arrays used for the `ListPreference` to include the title "Choose a delivery method." Use the same delivery choices that are used in the radio buttons in the `OrderActivity`.
3. Make the user's chosen delivery method appear in the same `Toast` message as the chosen **Market** setting.
4. Extra credit: Show the selected delivery method as the setting summary text that appears underneath the `ListPreference` title. Enable this text to change with each update.



Answer these questions

Question 1

In which file of the DroidCafeWithSettings project do you define the array of entries and the array of values for the `ListPreference`? Choose one:

- `pref_general.xml`
- `strings.xml`
- `menu_main.xml`
- `activity_main.xml`
- `content_main.xml`

Question 2

In which file of the DroidCafeWithSettings project do you *use* the array of entries and the array of values in setting up the `ListPreference`, and also set the `ListPreference` key and default value? Choose one:

- `pref_general.xml`
- `strings.xml`
- `menu_main.xml`
- `content_main.xml`
- `SettingsActivity.java`

Question 3

How do you set up a settings `Activity` and a `Fragment` with a `SwitchPreference` for the UI, and still remain compatible with the `v7 appcompat library` for backward compatibility with older versions of Android?

- Use a settings activity that extends `Activity`, a fragment that extends `PreferenceFragment`, and a `SwitchPreference` for the UI.
- Change `MainActivity` to extend `Activity`.
- Use a settings activity that extends `AppCompatActivity`, a fragment that extends `PreferenceFragmentCompat`, and a `SwitchPreferenceCompat` for the UI.
- You can't use a fragment with a `SwitchPreference` and remain compatible with the `v7 appcompat library`.

Submit your app for grading

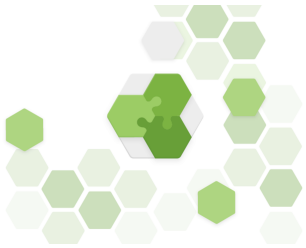
Guidance for graders

Check that the app has the following features:

- The `onCreate()` method reads the `deliveryPref` setting using `sharedPref.getString()`.
- The `pref_general.xml` file includes a `ListPreference` that uses for its entries an array of delivery choices.
- Extra credit: The statement `bindPreferenceSummaryToValue(findPreference("delivery"))` has been added to the `onCreate()` method of the `GeneralPreferenceFragment` class in the `SettingsActivity` in order to show the delivery choice in the preference summary.

Lesson 10.1 Part A: Room, LiveData, and ViewModel

Introduction



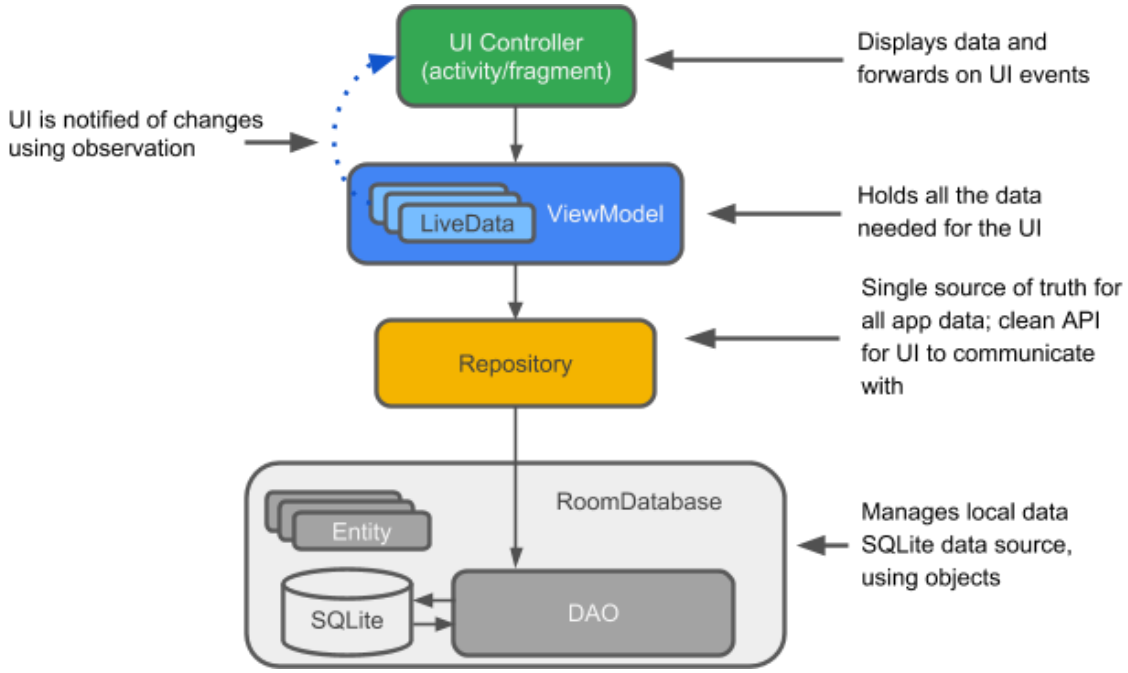
The Android operating system provides a strong foundation for building apps that run well on a wide range of devices and form factors. However, issues like complex lifecycles and the lack of a recommended app architecture make it challenging to write robust apps. The Android Architecture Components provide libraries for common tasks such as lifecycle management and data persistence to make it easier to implement the recommended architecture.

Architecture Components help you structure your app in a way that is robust, testable, and maintainable with less boilerplate code.

What are the recommended Architecture Components?

When it comes to architecture, it helps to see the big picture first. To introduce the terminology, here's a short overview of the Architecture Components and how they work together. Each component is explained more as you use it in this practical.

The diagram below shows a basic form of the recommended architecture for apps that use Architecture Components. The architecture consists of a UI controller, a `ViewModel` that serves `LiveData`, a Repository, and a Room database. The Room database is backed by an SQLite database and accessible through a data access object (DAO). Each component is described briefly below, and in detail in the Architecture Components concepts chapter, 10.1: Storing data with Room. You implement the components in this practical.



Because all the components interact, you will encounter references to these components throughout this practical, so here is a short explanation of each.

Entity: In the context of Architecture Components, the entity is an annotated class that describes a database table.

SQLite database: On the device, data is stored in an SQLite database. The Room persistence library creates and maintains this database for you.

DAO: Short for *data access object*. A mapping of SQL queries to functions. You used to have to define these queries in a helper class. When you use a DAO, your code calls the functions, and the components take care of the rest.

Room database: Database layer on top of an SQLite database that takes care of mundane tasks that you used to handle with a helper class. The Room database uses the DAO to issue queries to the SQLite database based on functions called.

Repository: A class that you create for managing multiple data sources. In addition to a Room database, the Repository could manage remote data sources such as a web server.

`ViewModel`*:* Provides data to the UI and acts as a communication center between the Repository and the UI. Hides the backend from the UI. `ViewModel` instances survive device configuration changes.

`LiveData`*:* A data holder class that follows the observer pattern, which means that it can be observed. Always holds/caches latest version of data. Notifies its observers when the data has changed. Generally, UI components observe relevant data. `LiveData` is lifecycle-aware, so it automatically manages stopping and resuming observation based on the state of its observing activity or fragment.

What you should already know

You should be able to create and run apps in Android Studio 3.0 or higher. In particular, be familiar with:

- `RecyclerView` and adapters
- SQLite databases and the SQLite query language
- Threading in general, and `AsyncTask` in particular

It helps to be familiar with:

- Software architectural patterns that separate data from the UI.

- The observer pattern. In summary, the observer pattern defines a one-to-many dependency between objects. Whenever an object changes its state, all the object's dependents are notified and updated automatically. The main object is called the “subject” and its dependents are called the "observers." Usually, the subject notifies the observers by calling one of the observers' methods. The subject knows what methods to call, because the observers are "registered" with the subject and specify the methods to call.

Important: This practical implements the architecture defined in the Guide to App Architecture and explained in the Architecture Components concepts chapter, 10.1: Storing data with Room. It is highly recommended that you read the concepts chapter.

What you'll learn

- How to design and construct an app using some of the Android Architecture Components. You'll use Room, `ViewModel`, and `LiveData`.

What you'll do

- Create an app with an `Activity` that displays words in a `RecyclerView`.
- Create an `Entity` that represents word objects.
- Define the mapping of SQL queries to Java methods in a DAO (data access object).
- Use `LiveData` to make changes to the data visible to the UI, by way of observers.
- Add a Room database to the app for persisting data locally, and initialize the database.
- Abstract the data backend as a `Repository` class with an API that is agnostic to how the data is stored or acquired.
- Use a `ViewModel` to separate all data operations from the UI.
- Add a second `Activity` that allows the user to add new words.

App overview

In this practical you build an app that uses the Android Architecture Components. The app, called RoomWordsSample, stores a list of words in a Room database and displays the list in a `RecyclerView`. The RoomWordsSample app is basic, but sufficiently complete that you can use it as a template to build on.

The RoomWordsSample app does the following:

- Works with a database to get and save words, and pre-populates the database with some words.
- Displays all the words in a `RecyclerView` in `MainActivity`.
- Opens a second `Activity` when the user taps the + FAB button. When the user enters a word, the app adds the word to the database and then the list updates automatically.

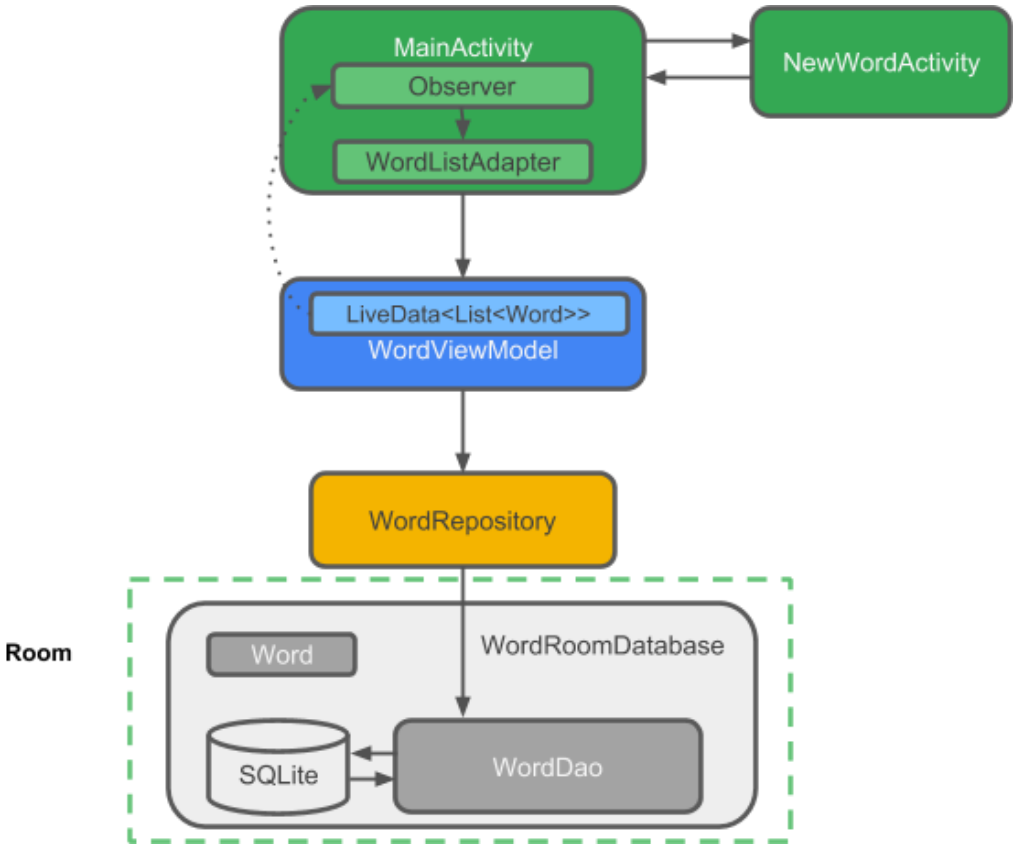
The screenshots below show the following:

- The RoomWordsSample app as it starts, with the initial list of words
- The activity to add a word

RoomWordsSample architecture overview

The following diagram mirrors the overview diagram from the introduction and shows all the pieces of the RoomWordsSample app. Each of the enclosing boxes (except for the SQLite database) represents a class that you create.

Tip: Print or open this diagram in a separate tab so you can refer to it as you build the code.



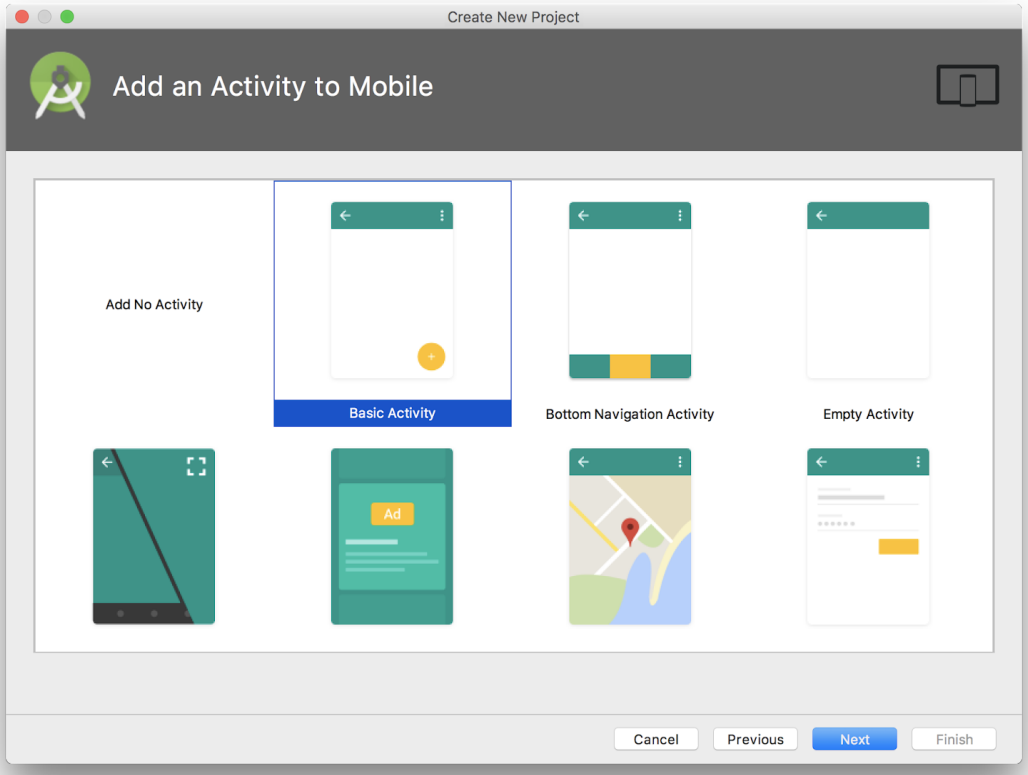
Task 1: Create the RoomWordsSample app

Note: In this practical, you are expected to create member variables, import classes, and extract values as needed. Code that you are expected to be familiar with is provided but not explained.

1.1 Create an app with one Activity

Open Android Studio and create an app. On the setup screens, do the following:

- Name the app RoomWordsSample.
- If you see check boxes for **Include Kotlin support** and **Include C++ support**, uncheck both boxes.
- Select only the **Phone & Tablet** form factor, and set the minimum SDK to API 14 or higher.
- Select the **Basic Activity**.



1.2 Update Gradle files

In Android Studio, manually add the Architecture Component libraries to your Gradle files.

1. Add the following code to your `build.gradle (Module: app)` file, to the bottom of the dependencies block (but still inside it).

```
// Room components
implementation
"android.arch.persistence.room:runtime:$rootProject.roomVersion"
annotationProcessor
"android.arch.persistence.room:compiler:$rootProject.roomVersion"
androidTestImplementation
"android.arch.persistence.room:testing:$rootProject.roomVersion"

// Lifecycle components
implementation
"android.arch.lifecycle:extensions:$rootProject.archLifecycleVersion"
annotationProcessor
"android.arch.lifecycle:compiler:$rootProject.archLifecycleVersion"
```

2. In your `build.gradle (Project: RoomWordsSample)` file, add the version numbers at the end of the file.

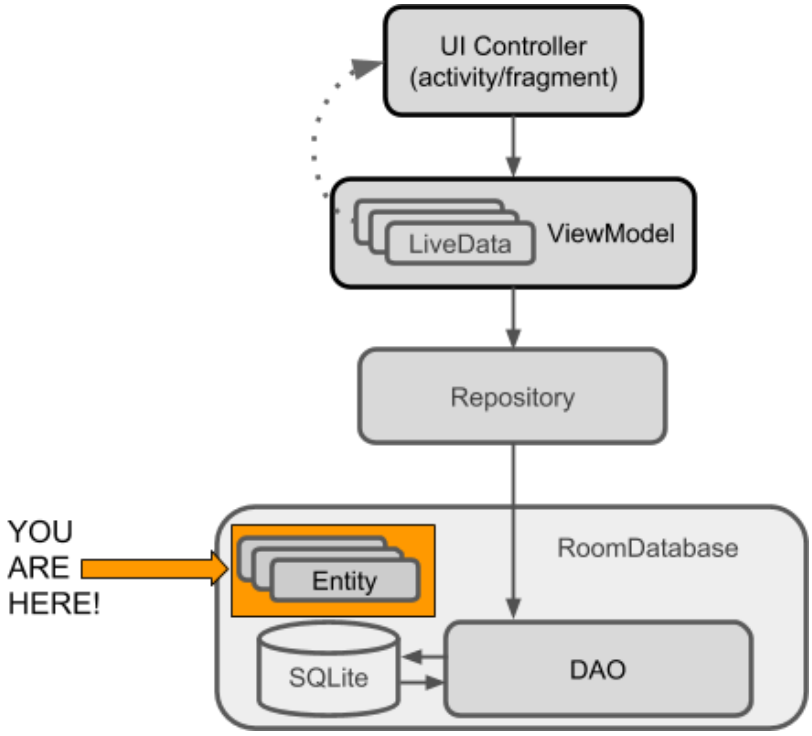
```
ext {

   roomVersion = '1.0.0'

   archLifecycleVersion = '1.1.0'

}
```

Important: Use the latest version numbers for the Room and lifecycle libraries. To find the latest version numbers:

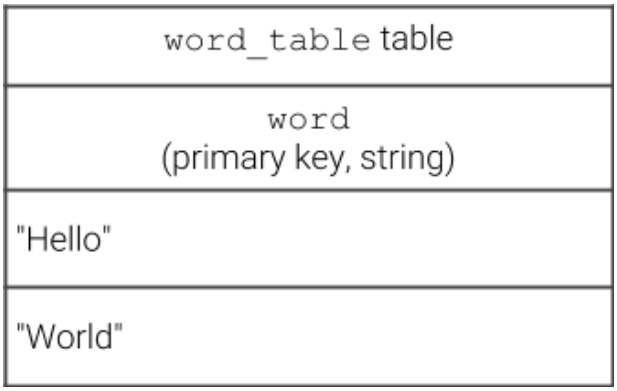
On the Adding Components to your Project page, find the entry for the component, for example Room.
The version number is defined at the start of the component's `dependencies` definition.
For example, the Room version number in the definition below is 1.1.1:
`def room_version = "1.1.1"`

Task 2: Create the Word entity

The diagram below is the complete architecture diagram with the component that you are going to implement in this task highlighted. Every task will have such a diagram to help you understand where the current component fits into the overall structure of the app, and to see how the components are connected.



The data for this app is words, and each word is represented by an entity in the database. In this task you create the `Word` class and annotate it so Room can create a database table from it. The diagram below shows a `word_table` database table. The table has one `word` column, which also acts as the primary key, and two rows, one each for "Hello" and "World."



2.1 Create the Word class

1. Create a class called `Word`.
2. Add a constructor that takes a `word` string as an argument. Add the `@NonNull` annotation so that the parameter can never be `null`.
3. Add a "getter" method called `getWord()` that returns the word. Room requires "getter" methods on the entity classes so that it can instantiate your objects.

```
public class Word {

    private String mWord;

    public Word(@NonNull String word) {this.mWord = word;}

    public String getWord(){return this.mWord;}

}
```

2.2 Annotate the Word class

To make the `Word` class meaningful to a Room database, you must annotate it. Annotations identify how each part of the `Word` class relates to an entry in the database. Room uses this information to generate code.

You use the following annotations in the steps below:

- `@Entity(tableName = "word_table")`
  Each `@Entity` class represents an entity in a table. Annotate your class declaration to indicate that the class is an entity. Specify the name of the table if you want it to be different from the name of the class.
- `@PrimaryKey`
  Every entity needs a primary key. To keep things simple, each word in the RoomWordsSample app acts as its own primary key. To learn how to auto-generate unique keys, see the tip below.
- `@NonNull`
  Denotes that a parameter, field, or method return value can never be `null`. The primary key should always use this annotation. Use this annotation for any mandatory fields in your rows.
- `@ColumnInfo(name = "word")`
  Specify the name of a column in the table, if you want the column name to be different from the name of the member variable.
- Every field that's stored in the database must either be public or have a "getter" method. This app provides a `getWord()` "getter" method rather than exposing member variables directly.

For a complete list of annotations, see the Room package summary reference.

Update your `Word` class with annotations, as shown in the code below.

1. Add the `@Entity` notation to the class declaration and set the `tableName` to `"word_table"`.
2. Annotate the `mWord` member variable as the `@PrimaryKey`. Require `mWord` to be `@NonNull`, and name the column `"word"`.

Note: If you *type in* the annotations, Android Studio auto-imports everything you need.

Here is the complete code:

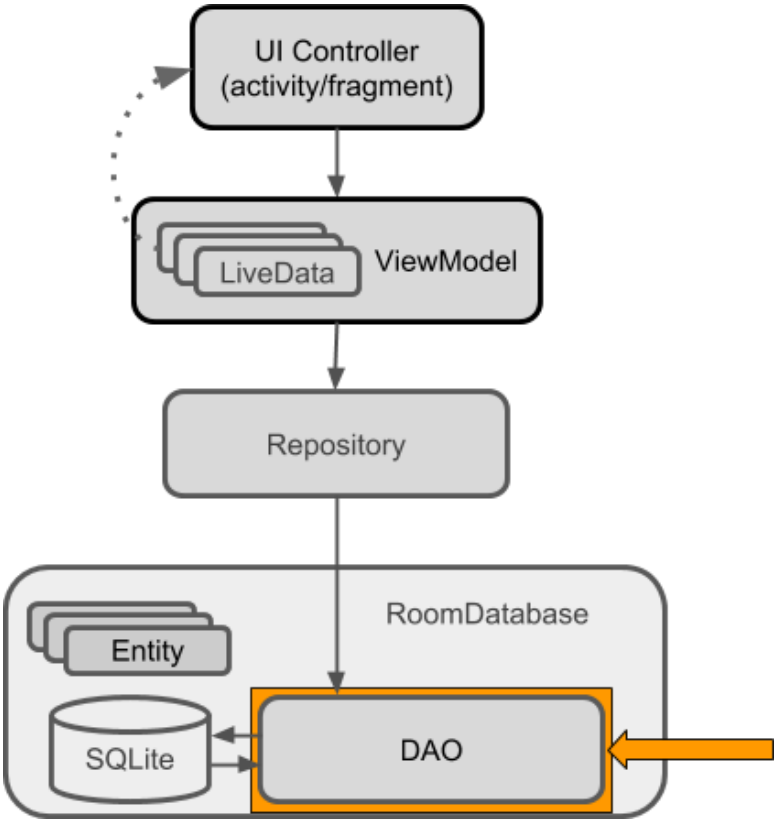
```
@Entity(tableName = "word_table")
public class Word {


    @PrimaryKey
    @NonNull
    @ColumnInfo(name = "word")
    private String mWord;


    public Word(@NonNull String word) {this.mWord = word;}


    public String getWord(){return this.mWord;}
}
```

If you get errors for the annotations, you can import them manually, as follows:

```
import android.arch.persistence.room.ColumnInfo;

import android.arch.persistence.room.Entity;

import android.arch.persistence.room.PrimaryKey;

import android.support.annotation.NonNull;
```

Tip on auto-generating keys: To auto-generate a unique key for each entity, you would add and annotate a primary integer key with `autoGenerate=true`. See Defining data using Room entities.

Task 3: Create the DAO



The data access object, or `Dao`, is an annotated class where you specify SQL queries and associate them with method calls. The compiler checks the SQL for errors, then generates queries from the annotations. For common queries, the libraries provide convenience annotations such as `@Insert`.

Note that:

- The DAO must be an `interface` or `abstract` class.
- Room uses the DAO to create a clean API for your code.
- By default, all queries (`@Query`) must be executed on a thread other than the main thread. (You work on that later.) For operations such as inserting or deleting, if you use the provided convenience annotations, Room takes care of thread management for you.

3.1 Implement the DAO class

The DAO for this practical is basic and only provides queries for getting all the words, inserting words, and deleting all the words.

1. Create a new `interface` and call it `WordDao`.
2. Annotate the class declaration with `@Dao` to identify the class as a DAO class for Room.
3. Declare a method to insert one word:

```
void insert(Word word);
```

4. Annotate the `insert()` method with `@Insert`. You don't have to provide any SQL! (There are also `@Delete` and `@Update` annotations for deleting and updating a row, but you do not use these operations in the initial version of this app.)
5. Declare a method to delete all the words:

```
void deleteAll();
```

6. There is no convenience annotation for deleting multiple entities, so annotate the `deleteAll()` method with the generic `@Query`. Provide the SQL query as a string parameter to `@Query`. Annotate the `deleteAll()` method as follows:

```
@Query("DELETE FROM word_table")
```

7. Create a method called `getAllWords()` that returns a `List` of `Words`:

```
List<Word> getAllWords();
```

8. Annotate the `getAllWords()` method with an SQL query that gets all the words from the `word_table`, sorted alphabetically for convenience:

```
@Query("SELECT * from word_table ORDER BY word ASC")
```

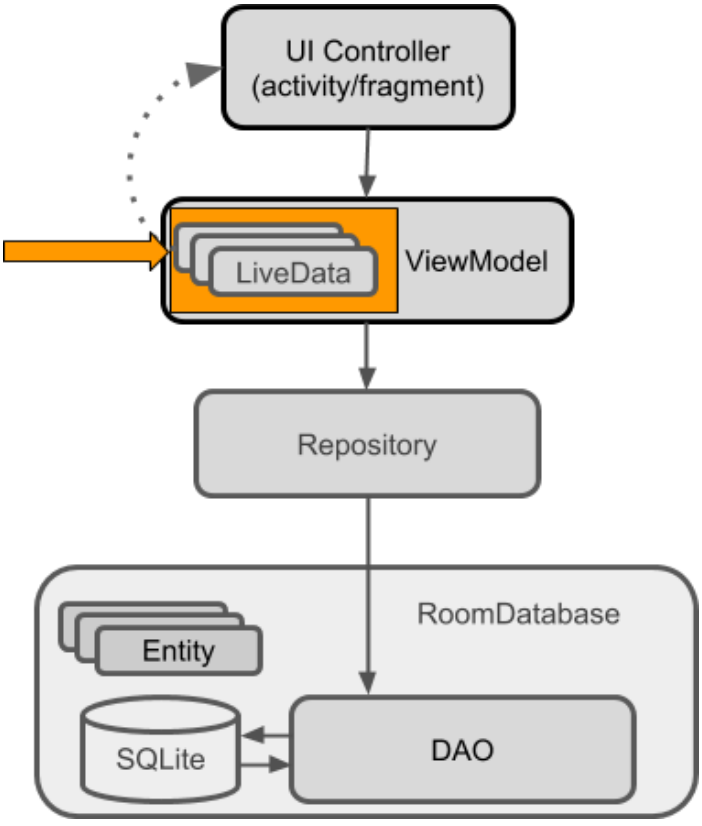
Here is the completed code for the `WordDao` class:

```
@Dao
public interface WordDao {

    @Insert
    void insert(Word word);

    @Query("DELETE FROM word_table")
    void deleteAll();

    @Query("SELECT * from word_table ORDER BY word ASC")
    List<Word> getAllWords();
}
```

Tip: For this app, ordering the words is not strictly necessary. However, by default, return order is not guaranteed, and ordering makes testing straightforward.

To learn more about DAOs, see Accessing data using Room DAOs.

Task 4: Use LiveData



When you display data or use data in other ways, you usually want to take some action when the data changes. This means you have to observe the data so that when it changes, you can react.

`LiveData`, which is a lifecycle library class for data observation, can help your app respond to data changes. If you use a return value of type `LiveData` in your method description, Room generates all necessary code to update the `LiveData` when the database is updated.

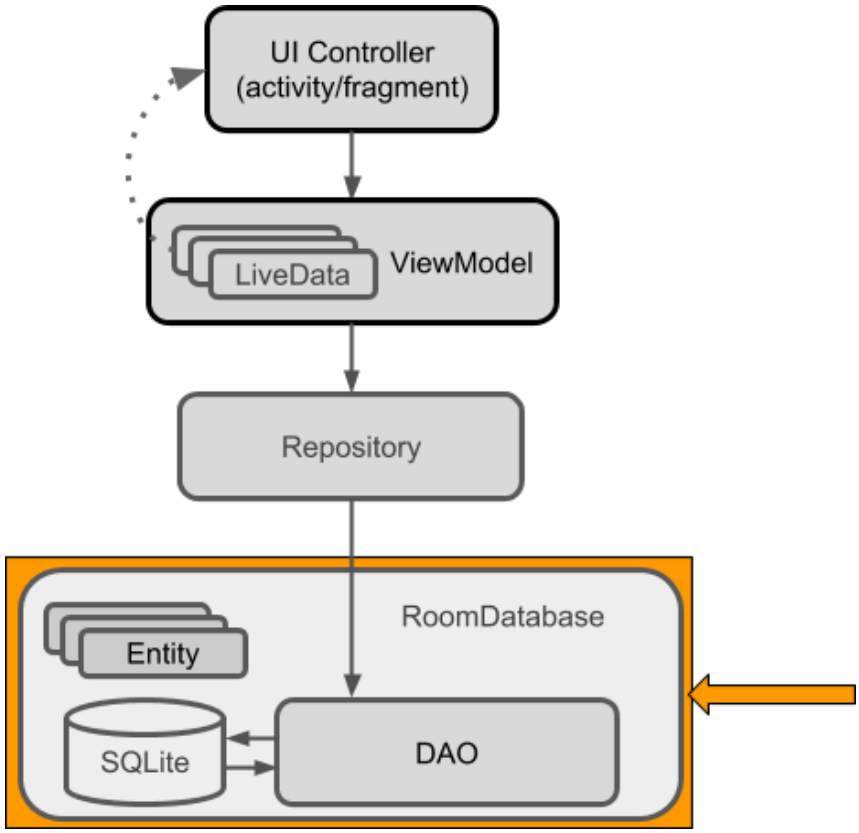
4.1 Return LiveData in WordDao

- In the `WordDao` interface, change the `getAllWords()` method signature so that the returned `List<Word>` is wrapped with `LiveData<>`.

```
@Query("SELECT * from word_table ORDER BY word ASC")

LiveData<List<Word>> getAllWords();
```

See the `LiveData` documentation to learn more about other ways to use `LiveData`, or watch this Architecture Components: LiveData and Lifecycle video.

Task 5: Add a Room database



Room is a database layer on top of an SQLite database. Room takes care of mundane tasks that you used to handle with a database helper class such as `SQLiteOpenHelper`.

- Room uses the DAO to issue queries to its database.
- By default, to avoid poor UI performance, Room doesn't allow you to issue database queries on the main thread. `LiveData` applies this rule by automatically running the query asynchronously on a background thread, when needed.

- Room provides compile-time checks of SQLite statements.
- Your `Room` class must be abstract and extend `RoomDatabase`.
- Usually, you only need one instance of the Room database for the whole app.

5.1 Implement a Room database

1. Create a `public abstract` class that extends `RoomDatabase` and call it `WordRoomDatabase`.

```
public abstract class WordRoomDatabase extends RoomDatabase {}
```

2. Annotate the class to be a Room database. Declare the entities that belong in the database—in this case there is only one entity, `Word`. (Listing the `entities` class or classes creates corresponding tables in the database.) Set the version number.

```
@Database(entities = {Word.class}, version = 1)
```

3. Define the DAOs that work with the database. Provide an abstract "getter" method for each `@Dao`.

```
public abstract WordDao wordDao();
```

4. Create the `WordRoomDatabase` as a singleton to prevent having multiple instances of the database opened at the same time, which would be a bad thing. Here is the code to create the singleton:

```
private static WordRoomDatabase INSTANCE;


public static WordRoomDatabase getDatabase(final Context context) {
   if (INSTANCE == null) {
        synchronized (WordRoomDatabase.class) {
            if (INSTANCE == null) {
                // Create database here
            }
        }
   }
   return INSTANCE;
}
```

5. Add code to create a database where indicated by the `Create database here` comment in the code above.

The following code uses Room's database builder to create a `RoomDatabase` object named `"word_database"` in the application context from the `WordRoomDatabase` class.

```
// Create database here
INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
        WordRoomDatabase.class, "word_database")
```

```
        .build();
```

6. Add a migration strategy for the database.

In this practical you don't update the entities and the version numbers. However, if you modify the database schema, you need to update the version number and define how to handle migrations. For a sample app such as the one you're creating, destroying and re-creating the database is a fine migration strategy. For a real app, you must implement a non-destructive migration strategy. See Understanding migrations with Room.

Add the following code to the builder, before calling `build()`

```
// Wipes and rebuilds instead of migrating
// if no Migration object.
// Migration is not part of this practical.
.fallbackToDestructiveMigration()
```

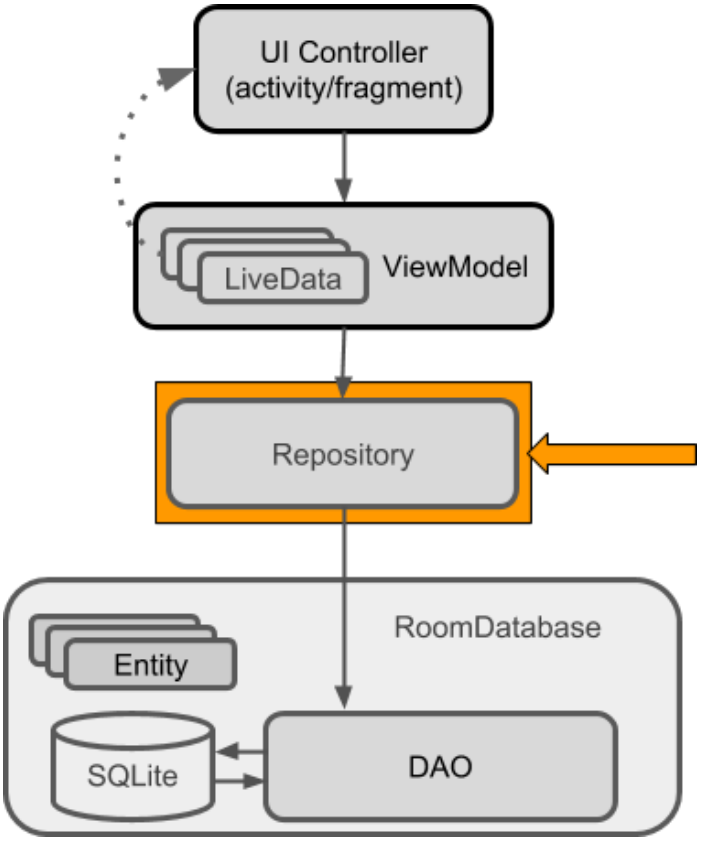
Here is the complete code for the whole `WordRoomDatabase` class:

```
@Database(entities = {Word.class}, version = 1)
public abstract class WordRoomDatabase extends RoomDatabase {

   public abstract WordDao wordDao();
   private static WordRoomDatabase INSTANCE;

   static WordRoomDatabase getDatabase(final Context context) {
       if (INSTANCE == null) {
           synchronized (WordRoomDatabase.class) {
               if (INSTANCE == null) {
                   INSTANCE =
Room.databaseBuilder(context.getApplicationContext(),
                           WordRoomDatabase.class, "word_database")
                             // Wipes and rebuilds instead of migrating
                             // if no Migration object.
                            // Migration is not part of this practical.
                           .fallbackToDestructiveMigration()
```

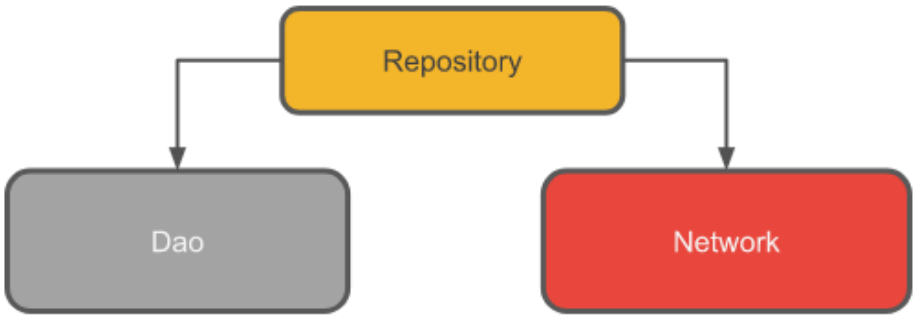
```
                            .build();
                }
            }
        }
        return INSTANCE;
    }
}
```

Important: In Android Studio, if you get errors when you paste code or during the build process, make sure you are using the full package name for imports. See Adding Components to your Project. Then select **Build > Clean Project**. Then select **Build > Rebuild Project**, and build again.

Task 6: Create the Repository



A *Repository* is a class that abstracts access to multiple data sources. The Repository is not part of the Architecture Components libraries, but is a suggested best practice for code separation and architecture. A `Repository` class handles data operations. It provides a clean API to the rest of the app for app data.



A Repository manages query threads and allows you to use multiple backends. In the most common example, the Repository implements the logic for deciding whether to fetch data from a network or use results cached in the local database.

6.1 Implement the Repository

1. Create a public class called `WordRepository`.
2. Add member variables for the DAO and the list of words.

```
private WordDao mWordDao;

private LiveData<List<Word>> mAllWords;
```

3. Add a constructor that gets a handle to the database and initializes the member variables.

```
WordRepository(Application application) {

    WordRoomDatabase db = WordRoomDatabase.getDatabase(application);

    mWordDao = db.wordDao();

    mAllWords = mWordDao.getAllWords();

}
```

4. Add a wrapper method called `getAllWords()` that returns the cached words as `LiveData`. Room executes all queries on a separate thread. Observed `LiveData` notifies the observer when the data changes.

```
LiveData<List<Word>> getAllWords() {

   return mAllWords;

}
```

5. Add a wrapper for the `insert()` method. Use an `AsyncTask` to call `insert()` on a non-UI thread, or your app will crash. Room ensures that you don't do any long-running operations on the main thread, which would block the UI.

```
public void insert (Word word) {

    new insertAsyncTask(mWordDao).execute(word);

}
```

6. Create the `insertAsyncTask` as an inner class. You should be familiar with `AsyncTask`, so here is the `insertAsyncTask` code for you to copy:

```
private static class insertAsyncTask extends AsyncTask<Word, Void, Void> {


    private WordDao mAsyncTaskDao;


    insertAsyncTask(WordDao dao) {
        mAsyncTaskDao = dao;
    }


    @Override
    protected Void doInBackground(final Word... params) {
        mAsyncTaskDao.insert(params[0]);
        return null;
    }
}
```

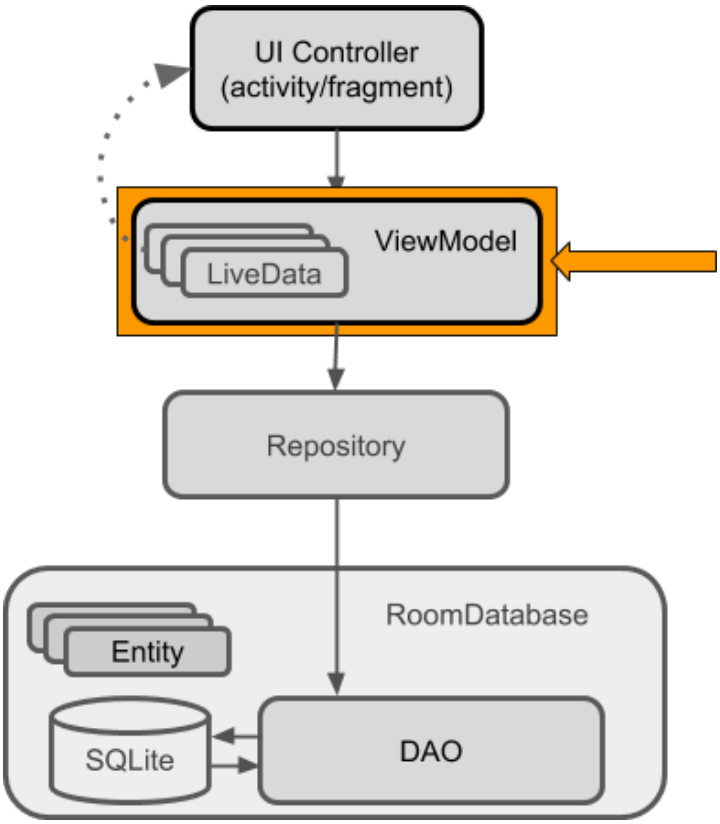
Here is the complete code for the `WordRepository` class:

```
public class WordRepository {


   private WordDao mWordDao;
   private LiveData<List<Word>> mAllWords;


   WordRepository(Application application) {
       WordRoomDatabase db = WordRoomDatabase.getDatabase(application);
       mWordDao = db.wordDao();
```

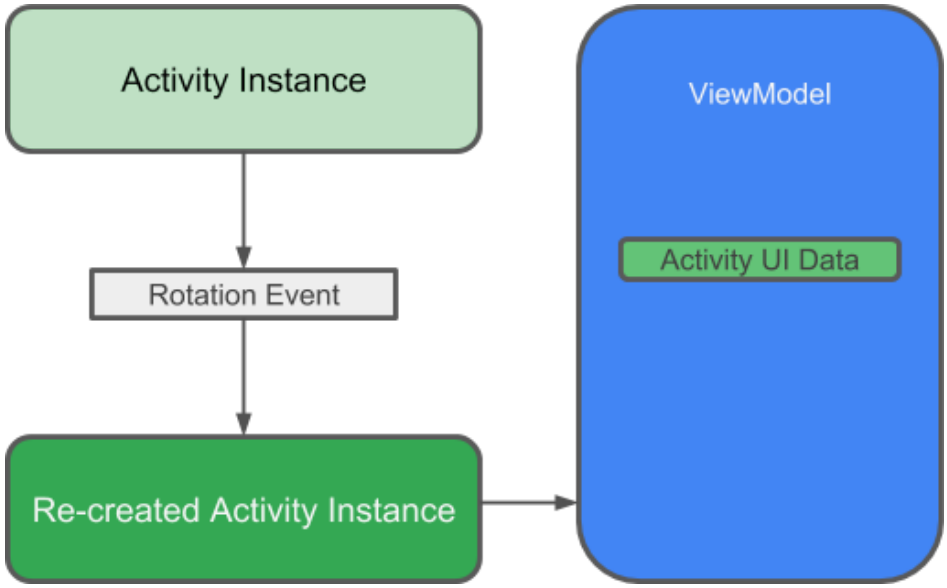
```
        mAllWords = mWordDao.getAllWords();
    }

    LiveData<List<Word>> getAllWords() {
        return mAllWords;
    }

    public void insert (Word word) {
        new insertAsyncTask(mWordDao).execute(word);
    }

    private static class insertAsyncTask extends AsyncTask<Word, Void, Void> {

        private WordDao mAsyncTaskDao;

        insertAsyncTask(WordDao dao) {
            mAsyncTaskDao = dao;
        }

        @Override
        protected Void doInBackground(final Word... params) {
            mAsyncTaskDao.insert(params[0]);
            return null;
        }
    }
}
```

Note: For this simple example, the Repository doesn't do much. For a more complex implementation, see the BasicSample code on GitHub.

Task 7: Create the ViewModel



The `ViewModel` is a class whose role is to provide data to the UI and survive configuration changes. A `ViewModel` acts as a communication center between the Repository and the UI. The `ViewModel` is part of the lifecycle library. For an introductory guide to this topic, see `ViewModel`.



A `ViewModel` holds your app's UI data in a way that survives configuration changes. Separating your app's UI data from your `Activity` and `Fragment` classes lets you better follow the single responsibility principle: Your activities and fragments are responsible for drawing data to the screen, while your `ViewModel` is responsible for holding and processing all the data needed for the UI.

In the `ViewModel`, use `LiveData` for changeable data that the UI will use or display.

7.1 Implement the WordViewModel

1. Create a class called `WordViewModel` that extends `AndroidViewModel`.

Warning:

Never pass context into `ViewModel` instances.
Do not store `Activity`, `Fragment`, or `View` instances or their `Context` in the `ViewModel`.

An `Activity` can be destroyed and created many times during the lifecycle of a `ViewModel`, such as when the device is rotated. If you store a reference to the `Activity` in the `ViewModel`, you end up with references that point to the destroyed `Activity`. This is a memory leak. If you need the application context, use `AndroidViewModel`, as shown in this practical.

```
public class WordViewModel extends AndroidViewModel {}
```

2. Add a private member variable to hold a reference to the Repository.

```
private WordRepository mRepository;
```

3. Add a private `LiveData` member variable to cache the list of words.

```
private LiveData<List<Word>> mAllWords;
```

4. Add a constructor that gets a reference to the `WordRepository` and gets the list of all words from the `WordRepository`.

```
public WordViewModel (Application application) {
    super(application);
    mRepository = new WordRepository(application);
    mAllWords = mRepository.getAllWords();
}
```

5. Add a "getter" method that gets all the words. This completely hides the implementation from the UI.

```
LiveData<List<Word>> getAllWords() { return mAllWords; }
```

6. Create a wrapper `insert()` method that calls the Repository's `insert()` method. In this way, the implementation of `insert()` is completely hidden from the UI.

```
public void insert(Word word) { mRepository.insert(word); }
```

Here is the complete code for `WordViewModel`:

```
public class WordViewModel extends AndroidViewModel {

    private WordRepository mRepository;

    private LiveData<List<Word>> mAllWords;

    public WordViewModel (Application application) {
        super(application);
        mRepository = new WordRepository(application);
        mAllWords = mRepository.getAllWords();
    }

    LiveData<List<Word>> getAllWords() { return mAllWords; }

    public void insert(Word word) { mRepository.insert(word); }
}
```

To learn more, watch the Architecture Components: `ViewModel` video.

Task 8: Add XML layouts for the UI

Next, add the XML layout for the list and items to be displayed in the `RecyclerView`.

This practical assumes that you are familiar with creating layouts in XML, so the code is just provided.

8.1 Add styles

1. Change the colors in `colors.xml` to the following: (to use Material Design colors):

```
<resources>
    <color name="colorPrimary">#2196F3</color>
    <color name="colorPrimaryLight">#64b5f6</color>
    <color name="colorPrimaryDark">#1976D2</color>
    <color name="colorAccent">#FFFF9800</color>
    <color name="colorTextPrimary">@android:color/white</color>
    <color name="colorScreenBackground">#fff3e0</color>
    <color name="colorTextHint">#E0E0E0</color>
</resources>
```

2. Add a style for text views in the `values/styles.xml` file:

```
<style name="text_view_style">
    <item name="android:layout_width">match_parent</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:textAppearance">
        @android:style/TextAppearance.Large</item>
    <item name="android:background">@color/colorPrimaryLight</item>
    <item name="android:layout_marginTop">8dp</item>
    <item name="android:layout_gravity">center</item>
    <item name="android:padding">16dp</item>
```

```
   <item name="android:textColor">@color/colorTextPrimary</item>
</style>
```

8.2 Add item layout

- Add a `layout/recyclerview_item.xml` layout:

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android=

      "http://schemas.android.com/apk/res/android"

   android:orientation="vertical" android:layout_width="match_parent"

   xmlns:tools="http://schemas.android.com/tools"

   android:layout_height="wrap_content">


   <TextView

       android:id="@+id/textView"

       android:layout_width="match_parent"

       android:layout_height="wrap_content"

       style="@style/text_view_style"

       tools:text="placeholder text" />

</LinearLayout>
```

8.3 Add the RecyclerView

1. In the `layout/content_main.xml` file, add a background color to the `ConstraintLayout`:

```
android:background="@color/colorScreenBackground"
```

2. In `content_main.xml` file, replace the `TextView` element with a `RecyclerView` element:

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/recyclerview"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    tools:listitem="@layout/recyclerview_item"
/>
```

8.4 Fix the icon in the FAB

The icon in your floating action button (FAB) should correspond to the available action. In the `layout/activity_main.xml` file, give the `FloatingActionButton` a `+` symbol icon:

1. Select **File > New > Vector Asset**.
2. Select **Material Icon**.
3. Click the Android robot icon in the **Icon:** field, then select the `+` ("add") asset.
4. In the `layout/activity_main.xml` file, in the `FloatingActionButton`, change the `srcCompat` attribute to:

```
android:src="@drawable/ic_add_black_24dp"
```

Task 9: Create an Adapter and adding the RecyclerView

You are going to display the data in a `RecyclerView`, which is a little nicer than just throwing the data in a `TextView`. This practical assumes that you know how `RecyclerView`, `RecyclerView.LayoutManager`, `RecyclerView.ViewHolder`, and `RecyclerView.Adapter` work.

9.1 Create the WordListAdapter class

- Add a class `WordListAdapter` that extends `RecyclerView.Adapter`. The adapter caches data and populates the `RecyclerView` with it. The inner class `WordViewHolder` holds and manages a view for one list item.

Here is the code:

```
public class WordListAdapter extends
RecyclerView.Adapter<WordListAdapter.WordViewHolder> {


   private final LayoutInflater mInflater;
```

```
private List<Word> mWords; // Cached copy of words


WordListAdapter(Context context) { mInflater = LayoutInflater.from(context); }


@Override
public WordViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View itemView = mInflater.inflate(R.layout.recyclerview_item, parent, false);
    return new WordViewHolder(itemView);
}


@Override
public void onBindViewHolder(WordViewHolder holder, int position) {
    if (mWords != null) {
        Word current = mWords.get(position);
        holder.wordItemView.setText(current.getWord());
    } else {
        // Covers the case of data not being ready yet.
        holder.wordItemView.setText("No Word");
    }
}


void setWords(List<Word> words){
    mWords = words;
    notifyDataSetChanged();
}


// getItemCount() is called many times, and when it is first called,
// mWords has not been updated (means initially, it's null, and we can't return
```

```
null).
    @Override
    public int getItemCount() {
        if (mWords != null)
            return mWords.size();
        else return 0;
    }

    class WordViewHolder extends RecyclerView.ViewHolder {
        private final TextView wordItemView;

        private WordViewHolder(View itemView) {
            super(itemView);
            wordItemView = itemView.findViewById(R.id.textView);
        }
    }
}
```

Note: The `mWords` variable in the adapter caches the data. In the next task, you add the code that updates the data automatically.

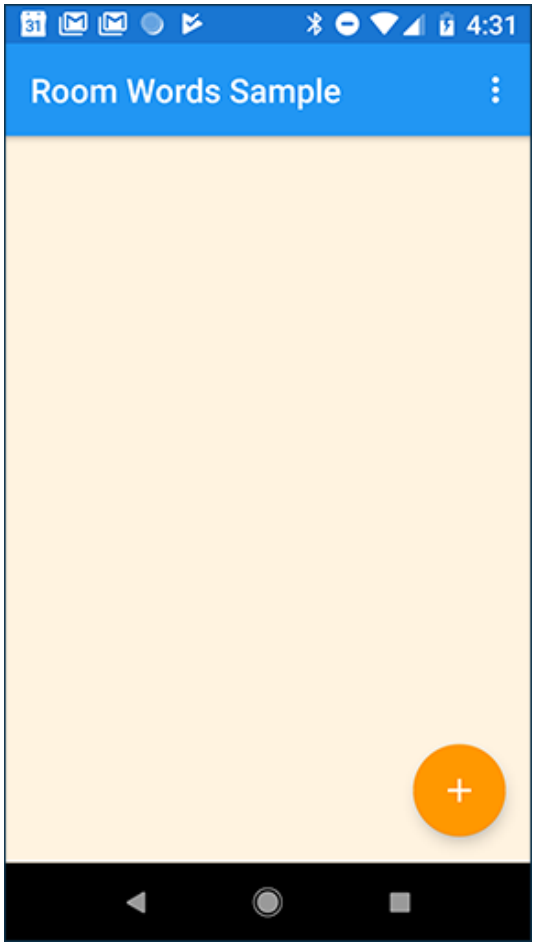
Note: The `getItemCount()` method needs to account gracefully for the possibility that the data is not yet ready and `mWords` is still `null`. In a more sophisticated app, you could display placeholder data or something else that would be meaningful to the user.

9.2 Add RecyclerView to MainActivity

1. Add the `RecyclerView` in the `onCreate()` method of `MainActivity`:

```
RecyclerView recyclerView = findViewById(R.id.recyclerview);

final WordListAdapter adapter = new WordListAdapter(this);

recyclerView.setAdapter(adapter);

recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

1. Run your app to make sure the app compiles and runs. There are no items, because you have not hooked up the data yet. The app should display the empty recycler view.



Task 10: Populate the database

There is no data in the database yet. You will add data in two ways: Add some data when the database is opened, and add an `Activity` for adding words. Every time the database is opened, all

content is deleted and repopulated. This is a reasonable solution for a sample app, where you usually want to restart on a clean slate.

10.1 Create the callback for populating the database

To delete all content and repopulate the database whenever the app is started, you create a `RoomDatabase.Callback` and override the `onOpen()` method. Because you cannot do Room database operations on the UI thread, `onOpen()` creates and executes an `AsyncTask` to add content to the database.

1. Add the `onOpen()` callback in the `WordRoomDatabase` class:

```
private static RoomDatabase.Callback sRoomDatabaseCallback =

    new RoomDatabase.Callback(){


    @Override

    public void onOpen (@NonNull SupportSQLiteDatabase db){

        super.onOpen(db);

       new PopulateDbAsync(INSTANCE).execute();

   }

};
```

1. Create an inner class `PopulateDbAsync` that extends `AsycTask`. Implement the `doInBackground()` method to delete all words, then create new ones. Here is the code for the `AsyncTask` that deletes the contents of the database, then populates it with an initial list of words. Feel free to use your own words!

```
/**
* Populate the database in the background.
```

```
*/
private static class PopulateDbAsync extends AsyncTask<Void, Void,
Void> {

   private final WordDao mDao;
   String[] words = {"dolphin", "crocodile", "cobra"};

   PopulateDbAsync(WordRoomDatabase db) {
       mDao = db.wordDao();
   }

   @Override
   protected Void doInBackground(final Void... params) {
       // Start the app with a clean database every time.
       // Not needed if you only populate the database
       // when it is first created
       mDao.deleteAll();

       for (int i = 0; i <= words.length - 1; i++) {
           Word word = new Word(words[i]);
           mDao.insert(word);
       }
       return null;
   }
}
```

1. Add the callback to the database build sequence in `WordRoomDatabase`, right before you call `.build()`:

```
.addCallback(sRoomDatabaseCallback)
```

Task 11: Connect the UI with the data

Now that you have created the method to populate the database with the initial set of words, the next step is to add the code to display those words in the `RecyclerView`.

To display the current contents of the database, you add an observer that observes the `LiveData` in the `ViewModel`. Whenever the data changes (including when it is initialized), the `onChanged()` callback is invoked. In this case, the `onChanged()` callback calls the adapter's `setWord()` method to update the adapter's cached data and refresh the displayed list.

11.1 Display the words

1. In `MainActivity`, create a member variable for the `ViewModel`, because all the activity's interactions are with the `WordViewModel` only.

```
private WordViewModel mWordViewModel;
```

1. In the `onCreate()` method, get a `ViewModel` from the `ViewModelProviders` class.

```
mWordViewModel = ViewModelProviders.of(this).get(WordViewModel.class);
```

Use `ViewModelProviders` to associate your `ViewModel` with your UI controller. When your app first starts, the `ViewModelProviders` class creates the `ViewModel`. When the activity is destroyed, for example through a configuration change, the `ViewModel` persists. When the activity is re-created, the `ViewModelProviders` return the existing `ViewModel`. See `ViewModel`.

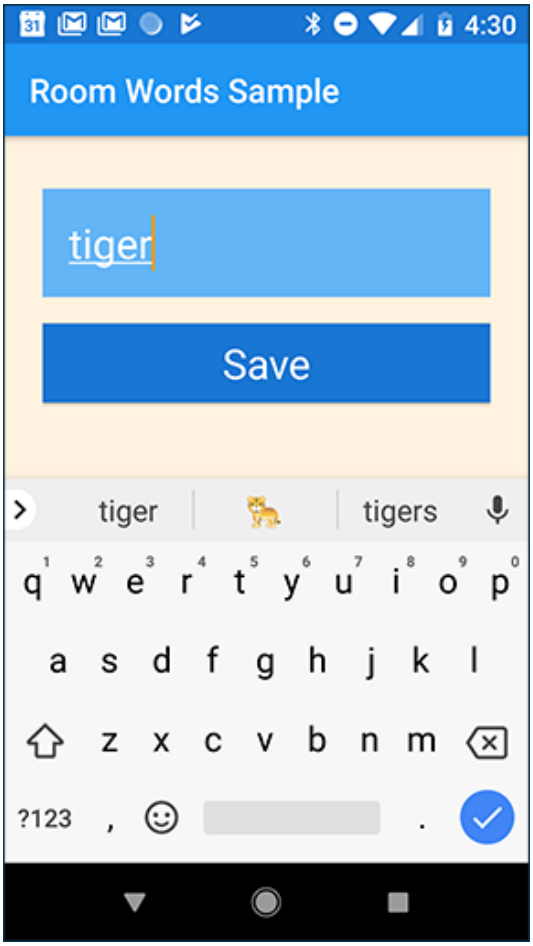
1. Also in `onCreate()`, add an observer for the `LiveData` returned by `getAllWords()`. When the observed data changes while the activity is in the foreground, the `onChanged()` method is invoked and updates the data cached in the adapter. Note that in this case, when the app opens, the initial data is added, so `onChanged()` method is called.

```
mWordViewModel.getAllWords().observe(this, new Observer<List<Word>>() {
   @Override
   public void onChanged(@Nullable final List<Word> words) {
       // Update the cached copy of the words in the adapter.
       adapter.setWords(words);
   }
});
```

2. Run the app. The initial set of words appears in the `RecyclerView`.

Task 12: Create an Activity for adding words

Now you will add an Activity that lets the user use the FAB to enter new words. This is what the interface for the new activity will look like:



12.1 Create the NewWordActivity

1. Add these string resources in the `values/strings.xml` file:

```
<string name="hint_word">Word…</string>

<string name="button_save">Save</string>
```

```
<string name="empty_not_saved">Word not saved because it is empty.</string>
```

1. Add a style for buttons in `value/styles.xml`:

```
<style name="button_style"
parent="android:style/Widget.Material.Button">
    <item name="android:layout_width">match_parent</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:background">@color/colorPrimaryDark</item>
    <item
name="android:textAppearance">@android:style/TextAppearance.Large</ite
m>
    <item name="android:layout_marginTop">16dp</item>
    <item name="android:textColor">@color/colorTextPrimary</item>
</style>
```

1. Use the Empty Activity template to create a new activity, `NewWordActivity`. Verify that the activity has been added to the Android Manifest.

```
<activity android:name=".NewWordActivity"></activity>
```

2. Update the `activity_new_word.xml` file in the layout folder:

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:background="@color/colorScreenBackground"

    android:orientation="vertical"

    android:padding="24dp">


    <EditText
```

```
        android:id="@+id/edit_word"
        style="@style/text_view_style"
        android:hint="@string/hint_word"
        android:inputType="textAutoComplete" />

    <Button
        android:id="@+id/button_save"
        style="@style/button_style"
        android:text="@string/button_save" />
</LinearLayout>
```

3. Implement the `NewWordActivity` class. The goal is that when the user presses the **Save** button, the new word is put in an `Intent` to be sent back to the parent `Activity`.

Here is the code for the `NewWordActivity` activity:

```
public class NewWordActivity extends AppCompatActivity {
   public static final String EXTRA_REPLY =
            "com.example.android.roomwordssample.REPLY";


   private  EditText mEditWordView;


   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_new_word);
       mEditWordView = findViewById(R.id.edit_word);


       final Button button = findViewById(R.id.button_save);
       button.setOnClickListener(new View.OnClickListener() {
           public void onClick(View view) {
               Intent replyIntent = new Intent();
               if (TextUtils.isEmpty(mEditWordView.getText())) {
                   setResult(RESULT_CANCELED, replyIntent);
               } else {
                   String word = mEditWordView.getText().toString();
                   replyIntent.putExtra(EXTRA_REPLY, word);
                   setResult(RESULT_OK, replyIntent);
               }
               finish();
           }
       });
   }
}
```

12.2 Add code to insert a word into the database

1. In `MainActivity`, add the `onActivityResult()` callback for the `NewWordActivity`. If the activity returns with `RESULT_OK`, insert the returned word into the database by calling the `insert()` method of the `WordViewModel`.

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {

   super.onActivityResult(requestCode, resultCode, data);


   if (requestCode == NEW_WORD_ACTIVITY_REQUEST_CODE && resultCode ==
RESULT_OK) {

       Word word = new Word(data.getStringExtra(NewWordActivity.EXTRA_REPLY));

       mWordViewModel.insert(word);

   } else {

       Toast.makeText(

               getApplicationContext(),

               R.string.empty_not_saved,

               Toast.LENGTH_LONG).show();

   }

}
```

1. Define the missing request code:

```
public static final int NEW_WORD_ACTIVITY_REQUEST_CODE = 1;
```

1. In `MainActivity`, start `NewWordActivity` when the user taps the FAB. Replace the code in the FAB's `onClick()` click handler with the following code:

```
Intent intent = new Intent(MainActivity.this, NewWordActivity.class);

startActivityForResult(intent, NEW_WORD_ACTIVITY_REQUEST_CODE);
```

1. Run your app. When you add a word to the database in `NewWordActivity`, the UI automatically updates.
2. Add a word that already exists in the list. What happens? Does your app crash?
   Your app uses the word itself as the primary key, and each primary key **must** be unique. You can specify a conflict strategy to tell your app what to do when the user tries to add an existing word.
3. In the `WordDao` interface, change the annotation for the `insert()` method to:

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
```

To learn about other conflict strategies, see the `OnConflictStrategy` reference.

4. Run your app again and try adding a word that already exists. What happens now?

Solution code

Android Studio project: RoomWordsSample

Summary

Now that you have a working app, let's recap what you've built. Here is the app structure again, from the beginning:

- You have an app that displays words in a list (`MainActivity`, `RecyclerView`, `WordListAdapter`).
- You can add words to the list (`NewWordActivity`).
- A word is an instance of the `Word` entity class.
- The words are cached in the `RecyclerViewAdapter` as a `List` of words (`mWords`). The list is automatically updated and redisplayed when the data changes.
- The automatic update happens because in the `MainActivity`, there is an `Observer` that observes the words and is notified when the words change. When there is a change, the observer's `onChange()` method is executed and updates `mWords` in the `WordListAdapter`.
- The data can be observed because it is `LiveData`. And what is observed is the `LiveData<List<Word>>` that is returned by the `WordViewModel` object.
- The `WordViewModel` hides everything about the backend from the user interface. It provides methods for accessing the UI data, and it returns `LiveData` so that `MainActivity` can set up the observer relationship. Views, activities, and fragments only interact with the data through the `ViewModel`. As such, it doesn't matter where the data comes from.
- In this case, the data comes from a Repository. The `ViewModel` does not need to know what that Repository interacts with. It just needs to know how to interact with the Repository, which is through the methods exposed by the Repository.

- The Repository manages one or more data sources. In the RoomWordsSample app, that backend is a Room database. Room is a wrapper around and implements an SQLite database. Room does a lot of work for you that you used to have to do yourself. For example, Room does everything that you used to use an `SQLiteOpenHelper` class to do.

- The DAO maps method calls to database queries, so that when the Repository calls a method such as `getAllWords()`, Room can execute `SELECT * from word_table ORDER BY word ASC`.

- The result returned from the query is observed `LiveData`. Therefore, every time the data in Room changes, the `Observer` interface's `onChanged()` method is executed and the UI is updated.

Related concept

The related concept documentation is in 10.1: Room, LiveData, and ViewModel.

Learn more

To continue working with the RoomWordsSample app and learn more ways to use a Room database, see the 10.1 Part B: Room, LiveData, and ViewModel codelab, which takes up where this codelab leaves off.

Android developer documentation:

- Guide to App Architecture
- Adding Components to your Project

- DAO
- Room DAOs
- Room package summary reference
- Handling Lifecycles with Lifecycle-Aware Components
- `LiveData`
- `MutableLiveData`
- `ViewModel`
- `ViewModelProviders`
- Defining data using Room entities

Blogs and articles:

- 7 Steps To Room (migrating an existing app)
- Understanding migrations with Room
- Lifecycle Aware Data Loading with Architecture Components

Codelabs:

- Android Persistence codelab (`LiveData`, Room, DAO)
- Android lifecycle-aware components codelab (`ViewModel`, `LiveData`, `LifecycleOwner`, `LifecycleRegistryOwner`)

Videos:

- Architecture Components overview
- Architecture Components: LiveData and Lifecycle
- Architecture Components: ViewModel

Code samples:

- Architecture Components code samples
- BasicSample (a not-so-basic but comprehensive sample)

Homework

Build and run an app

Create an app that uses a Room database, `ViewModel`, and `LiveData` to display the data when the data changes. You can make this as simple or as sophisticated as you wish, as long as the app uses all the required components, and the data updates on the screen when the data changes in the database.

Here are some hints and ideas:

- Create a simple app that stores one text document and displays the contents of the document in a `TextView`. When the user edits the document, changes appear in the `TextView`.
- Create a question-answer app. Start with only questions and let users add new questions and answers.
- As a challenge, add a button to each answer in the question-answer app that displays additional information that's stored in a different repository. The information could come from a file on the device, or from a page on the internet.

Answer these questions

Question 1

What are the advantages of using a Room database?

- Creates and manages an Android SQLite database for you.
- Eliminates a lot of boilerplate code.
- Helps you manage multiple backends.
- Using a DAO, provides a mechanism for mapping Java methods to database queries.

Question 2

Which of the following are reasons for using a `ViewModel`?

- Cleanly separates the UI from the backend.
- Often used with `LiveData` for changeable data that the UI will use or display.
- Prevents your data from being lost when the app crashes.
- Acts as a communication center between the Repository and the UI.
- `ViewModel` instances survive device configuration changes.

Question 3

What is the DAO?

- Short for "data access object."
- A library for managing database queries.
- An annotated interface that maps Java methods to SQLite queries.
- A class whose methods run always in the background, not on the main thread.
- A class that the compiler checks for SQL errors, then uses to generate queries from the annotations.

Question 4

What are features of `LiveData`?

- When `LiveData` is used with Room, data updates automatically if all the intermediate levels also return `LiveData` (DAO, `ViewModel`, Repository).
- Uses the observer pattern and notifies its observers when its data has changed.
- Automatically updates the UI when it changes.
- Is lifecycle aware.

Submit your app for grading

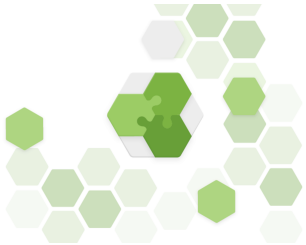
Guidance for graders

Check that the app has the following features:

- Uses the Room database to store data.
- Uses `ViewModel` and `LiveData` to manage and display changing data.
- Has a way to edit data and show changes.

Lesson 10.1 Part B: Room, LiveData, and ViewModel

Introduction



This codelab (practical) follows on from 10.1 Part A: Room, LiveData, and ViewModel. This codelab gives you more practice at using the API provided by the Room library to implement database functionality. You will add the ability to delete specific items from the database. This codelab also includes a coding challenge, in which you update the app so the user can edit existing data.

What you should already know

You should be able to create and run apps in Android Studio 3.0 or higher. In particular, be familiar with the following:

- Using `RecyclerView` and adapters.

- Using entity classes, data access objects (DAOs), and the `RoomDatabase` to store and retrieve data in Android's built-in SQLite database. You learned these topics in 10.1 Part A: Room, LiveData, and ViewModel.

What you'll learn

- How to populate the database with data only if the database is empty (so users don't lose changes they made to the data).
- How to delete data from a Room database.
- How to update existing data (if you build the challenge app).

What you'll do

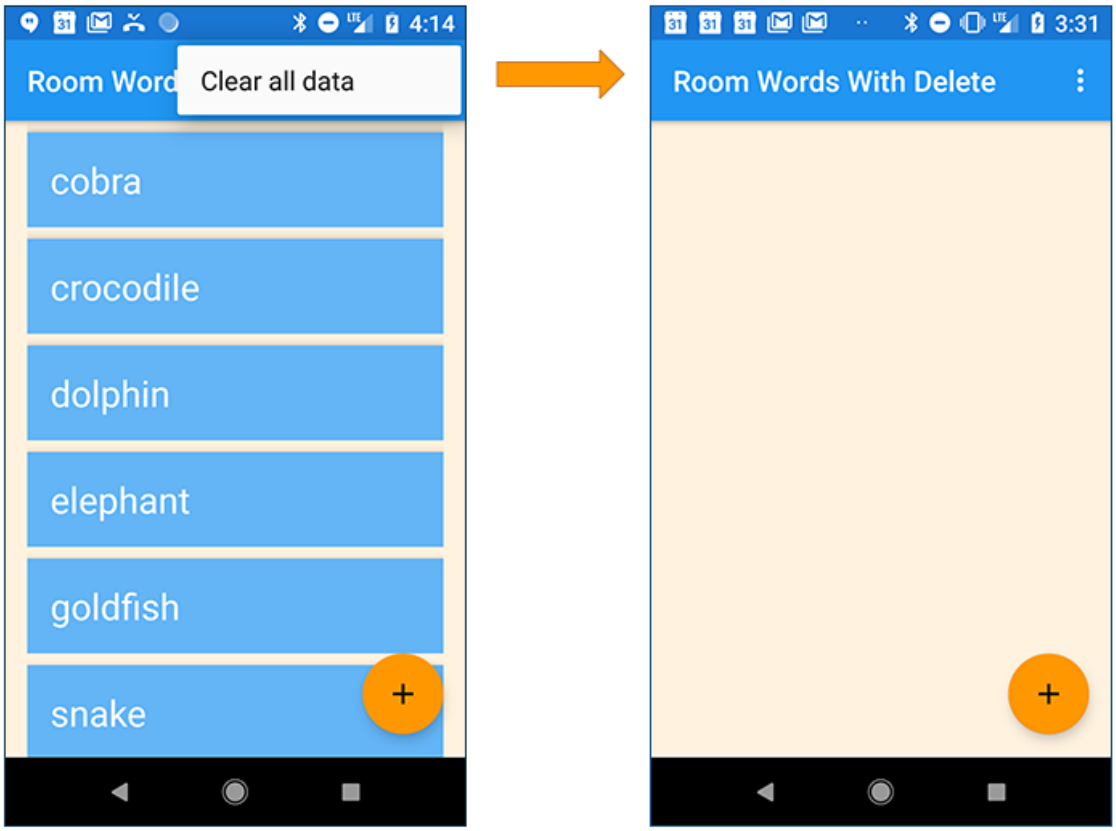
- Update the RoomWordsSample app to keep data when the app closes.
- Allow users to delete all words by selecting an **Options** menu item.
- Allow users to delete a specific word by swiping an item in the list.
- Optionally, in a coding challenge, extend the app to allow the user to update existing words.

App overview

You will extend the RoomWordsSample app that you created in the previous codelab. So far, that app displays a list of words, and users can add words. When the app closes and re-opens, the app re-initializes the database. Words that the user has added are lost.

In this practical, you extend the app so that it only initializes the data in the database if there is no existing data.

Then you add a menu item that allows the user to delete all the data.



You also enable the user to swipe a word to delete it from the database.

Task 1: Initialize data only if the database is empty

The RoomWordsSample app that you created in the previous practical deletes and re-creates the data whenever the user opens the app. This behavior isn't ideal, because users will want their added words to remain in the database when the app is closed. (Solution code for the previous practical is in GitHub.)

In this task you update the app so that when it opens, the initial data set is only added if the database has no data.

To detect whether the database contains data already, you can run a query to get one data item. If the query returns nothing, then the database is empty.

Note: In a production app, you might want to allow users to delete all data without re-initializing the data when the app restarts. But for testing purposes, it's useful to be able to delete all data, then re-initialize the data when the app starts.

1.1 Add a method to the DAO to get a single word

Currently, the `WordDao` interface has a method for getting all the words, but not for getting any single word. The method to get a single word does not need to return `LiveData`, because your app will call the method explicitly when needed.

1. In the `WordDao` interface, add a method to get any word:

```
@Query("SELECT * from word_table LIMIT 1")
   Word[] getAnyWord();
```

`Room` issues the database query when the `getAnyWord()` method is called and returns an array containing one word. You don't need to write any additional code to implement it.

1.2 Update the initialization method to check whether data exists

Use the `getAnyWord()` method in the method that initializes the database. If there is any data, leave the data as it is. If there is no data, add the initial data set.

1. `PopulateDBAsync` is an inner class in `WordRoomDatbase`. In `PopulateDBAsync`, update the `doInBackground()` method to check whether the database has any words before initializing the data:

```
@Override
protected Void doInBackground(final Void... params) {

       // If we have no words, then create the initial list of words
       if (mDao.getAnyWord().length < 1) {
           for (int i = 0; i <= words.length - 1; i++) {
```

```
                Word word = new Word(words[i]);
                mDao.insert(word);
            }
        }
    return null;
}
```

2. Run your app and add several new words. Close the app and restart it. You should see the new words that you added, as the words should now persist when the app is closed and opened again.

Task 2: Delete all words

In the previous practical, you used the `deleteAll()` method to clear out all the data when the database opened. The `deleteAll()` method was only invoked from the `PopulateDbAsync` class when the app started. You will now make the `deleteAll()` method available through the `ViewModel` so that your app can call the method whenever it's needed.

Here are the general steps for implementing a method to use the `Room` library to interact with the database:

- Add the method to the DAO, and annotate it with the relevant database operation. For the `deleteAll()` method, you already did this step in the previous practical.
- Add the method to the `WordRepository` class. Write the code to run the method in the background.
- To call the method in the `WordRepository` class, add the method to the `WordViewModel`. The rest of the app can then access the method through the `WordViewModel`.

2.1 Add deleteAll() to the WordDao interface and annotate it

1. In `WordDao`, check that the `deleteAll()` method is defined and annotated with the SQL that runs when the method executes:

```
@Query("DELETE FROM word_table")
```

```
void deleteAll();
```

2.2 Add deleteAll() to the WordRepository class

Add the `deleteAll()` method to the `WordRepository` and implement an `AsyncTask` to delete all words in the background.

1. In `WordRepository`, define `deleteAllWordsAsyncTask` as an inner class. Implement `doInBackground()` to delete all the words by calling `deleteAll()` on the DAO:

```
private static class deleteAllWordsAsyncTask extends
AsyncTask<Void, Void, Void> {
    private WordDao mAsyncTaskDao;

    deleteAllWordsAsyncTask(WordDao dao) {
        mAsyncTaskDao = dao;
    }

    @Override
    protected Void doInBackground(Void... voids) {
        mAsyncTaskDao.deleteAll();
        return null;
    }
}
```

2. In the `WordRepository` class, add the `deleteAll()` method to invoke the `AsyncTask` that you defined.

```
public void deleteAll()  {
    new deleteAllWordsAsyncTask(mWordDao).execute();
}
```

2.3 Add deleteAll() to the WordViewModel class

Make the `deleteAll()` method available to the `MainActivity` by adding it to the `WordViewModel`.

1. In the `WordViewModel` class, add the `deleteAll()` method:

```
public void deleteAll() {mRepository.deleteAll();}
```

Task 3: Add an Options menu item to delete all data

Next, you add a menu item to enable users to invoke `deleteAll()`.

Note: The production version of your app must provide safeguards so that users do not accidentally wipe out their entire database. However, while you develop your app, it's helpful to be able to clear out test data quickly. This is especially true now that your app does not clear out the data when the app opens.

3.1 Add the Clear all data menu option

1. In `menu_main.xml`, change the menu option `title` and `id`, as follows:

```
<item
   android:id="@+id/clear_data"
   android:orderInCategory="100"
   android:title="@string/clear_all_data"
   app:showAsAction="never" />
```

2. In `MainActivity`, implement the `onOptionsItemSelected()` method to invoke the `deleteAll()` method on the `WordViewModel` object.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();

    if (id == R.id.clear_data) {
        // Add a toast just for confirmation
        Toast.makeText(this, "Clearing the data...",
                Toast.LENGTH_SHORT).show();

        // Delete the existing data
        mWordViewModel.deleteAll();
        return true;
    }

    return super.onOptionsItemSelected(item);
}
```

3. Run your app. In the **Options** menu, select **Clear all data**. All words should disappear.
4. Restart the app. (Restart it from your device or the emulator; don't run it again from Android Studio) You should see the initial set of words.

Note: After you clear the data, re-deploying the app from Android Studio shows the initial data set again. Opening the app shows the empty data set.

Task 4: Delete a single word

Your app lets users add words and delete all words. In Tasks 4 and 5, you extend the app so that users can delete a word by swiping the item in the `RecyclerView`.

Again, here are the general steps to implement a method to use the `Room` library to interact with the database:

- Add the method to the DAO, and annotate it with the relevant database operation.
- Add the method to the `WordRepository` class. Write the code to run the method in the background.
- To call the method in the `WordRepository` class, add the method to the `WordViewModel`. The rest of the app can then access the method through the `WordViewModel`.

4.1 Add deleteWord() to the DAO and annotate it

1. In `WordDao`, add the `deleteWord()` method:

```
@Delete
void deleteWord(Word word);
```

Because this operation deletes a single row, the `@Delete` annotation is all that is needed to delete the word from the database.

4.2 Add deleteWord() to the WordRepository class

1. In `WordRepository`, define another `AsyncTask` called `deleteWordAsyncTask` as an inner class. Implement `doInBackground()` to delete a word by calling `deleteWord()` on the DAO:

```
private static class deleteWordAsyncTask extends AsyncTask<Word, Void,
Void> {
   private WordDao mAsyncTaskDao;

   deleteWordAsyncTask(WordDao dao) {
       mAsyncTaskDao = dao;
   }

   @Override
   protected Void doInBackground(final Word... params) {
       mAsyncTaskDao.deleteWord(params[0]);
       return null;
```

```
    }
}
```

2. In `WordRepository`, add the `deleteWord()` method to invoke the `AsyncTask` you defined.

```
public void deleteWord(Word word)  {
    new deleteWordAsyncTask(mWordDao).execute(word);
}
```

4.3 Add deleteWord() to the WordViewModel class

To make the `deleteWord()` method available to other classes in the app, in particular, `MainActivity`, add it to `WordViewModel`.

1. In `WordViewModel`, add the `deleteWord()` method:

```
public void deleteWord(Word word) {mRepository.deleteWord(word);}
```

You have now implemented the logic to delete a word. As yet, there is no way to invoke the delete-word operation from the app's UI. You fix that next.

Task 5: Enable users to swipe words away

In this task, you add functionality to allow users to swipe an item in the `RecyclerView` to delete it.

Use the `ItemTouchHelper` class provided by the Android Support Library (version 7 and higher) to implement swipe functionality in your `RecyclerView`. The `ItemTouchHelper` class has the following methods:

- `onMove()` is called when the user moves the item. You will not implement any move functionality in this app.

- `onSwipe()` is called when the user swipes the item. You implement this method to delete the word that was swiped.

5.1 Enable the adapter to detect the swiped word

1. In `WordListAdapter`, add a method to get the word at a given position.

```
public Word getWordAtPosition (int position) {
   return mWords.get(position);
}
```

2. In `MainActivity`, in `onCreate()`, create the `ItemTouchHelper`. Attach the `ItemTouchHelper` to the `RecyclerView`.

```
// Add the functionality to swipe items in the
// recycler view to delete that item
ItemTouchHelper helper = new ItemTouchHelper(
    new ItemTouchHelper.SimpleCallback(0,
        ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT) {
           @Override
           public boolean onMove(RecyclerView recyclerView,
                                 RecyclerView.ViewHolder viewHolder,
                                 RecyclerView.ViewHolder target) {
               return false;
           }

           @Override
           public void onSwiped(RecyclerView.ViewHolder viewHolder,
                                int direction) {
               int position = viewHolder.getAdapterPosition();
               Word myWord = adapter.getWordAtPosition(position);
               Toast.makeText(MainActivity.this, "Deleting " +
                       myWord.getWord(), Toast.LENGTH_LONG).show();

               // Delete the word
               mWordViewModel.deleteWord(myWord);
            }
       });

helper.attachToRecyclerView(recyclerView);
```

.

Things to notice in the code:

`onSwiped()` gets the position of the `ViewHolder` that was swiped:

```
int position = viewHolder.getAdapterPosition();
```

Given the position, you can get the word displayed by the `ViewHolder` by calling the `getWordAtPosition()` method that you defined in the adapter:

```
Word myWord = adapter.getWordAtPosition(position);
```

Delete the word by calling `deleteWord()` on the `WordViewModel`:

```
mWordViewModel.deleteWord(myWord);
```

Now run your app and delete some words

1. Run your app. You should be able to delete words by swiping them left or right.

Solution code

Android Studio project: RoomWordsWithDelete

Coding challenge

Note: All coding challenges are optional and are not prerequisites for later lessons.

Challenge: Update your app to allow users to edit a word by tapping the word and then saving their changes.

Hints

Make changes in NewWordActivity

You can add functionality to `NewWordActivity`, so that it can be used either to add a new word or edit an existing word.

Use an auto-generated key in Word

The `Word` entity class uses the `word` field as the database key. However, when you update a row in the database, the item being updated cannot be the primary key, because the primary key is unique to each row and never changes. So you need to add an auto-generated `id` to use as the primary key

```
@PrimaryKey(autoGenerate = true)
private int id;

@NonNull
@ColumnInfo(name = "word")
private String mWord;
```

Add a constructor for `Word` that takes an `id`

Add a constructor to the `Word` entity class that takes `id` and `word` as parameters. Make sure this additional constructor is annotated using `@Ignore`, because `Room` expects only one constructor by default in an entity class.

```
@Ignore
public Word(int id, @NonNull String word) {
    this.id = id;
    this.mWord = word;
}
```

To update an existing `Word`, create the `Word` using this constructor. `Room` will use the primary key (in this case the `id`) to find the existing entry in the database so it can be updated.

In `WordDao`, add the `update()` method like this:

```
@Update
void update(Word... word);
```

Challenge solution code

Android Studio project: RoomWordsWithUpdate

Summary

Writing database code

- Room takes care of opening and closing the database connections each time a database operations executes.
- Annotate methods in the data access object (DAO) as `@insert`, `@delete`, `@update`, `@query`.
- For simple insertions, updates and deletions, it is enough to add the relevant annotation to the method in the DAO.

  For example:

```
@Delete
void deleteWord(Word word);

@Update
void update(Word... word);
```

- For queries or more complex database interactions, such as deleting all words, use the `@query` annotation and provide the SQL for the operation.

  For example:

```
@Query("SELECT * from word_table ORDER BY word ASC")
LiveData<List<Word>> getAllWords();

@Query("DELETE FROM word_table")
void deleteAll();
```

ItemTouchHelper

- To enable users to swipe or move items in a `RecyclerView`, you can use the `ItemTouchHelper` class.
- Implement `onMove()` and `onSwipe()`.
- To identify which item the user moved or swiped, you can add a method to the adapter for the `RecylerView`. The method takes a position and returns the relevant item. Call the method inside `onMove()` or `onSwipe()`.

Related concept

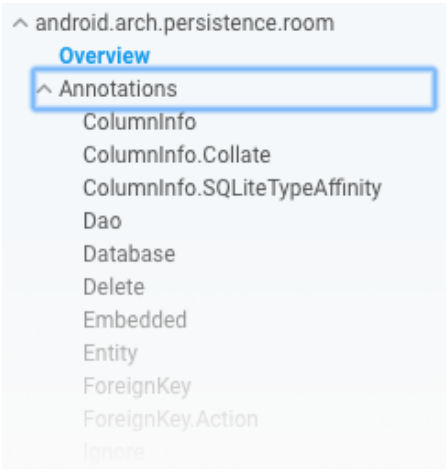
The related concept documentation is in 10.1: Room, LiveData, and ViewModel.

Learn more

Entities, data access objects (DAOs), and `ViewModel`:

- Defining data using Room entities
- Accessing data using Room DAOs
- `ViewModel` guide
- `Dao` reference
- `ViewModel` reference
- To see all the possible annotations for an entity, go to `android.arch.persistence.room` in the Android reference and expand the

Annotations menu item in the left nav.



`ItemTouchHelper`:

- `ItemTouchHelper` reference

Homework

Build and run an app

You've learned several ways to store data. Choosing the right storage option depends on how large your data is, and how long the data needs to survive.

Create an app that demonstrates how data that is stored in at least two different locations survives configuration changes and the destruction of the app. You can do this by storing small pieces of data, such as strings, in different data stores.

- The app should demonstrate what happens to data that is not saved.
- The app could demonstrate what happens to data that is preserved with `savedInstanceState`, data that uses `LiveData` with a `ViewModel`, and data that is stored in a file or database.

Answer these questions

Question 1

Android Architecture Components provide some convenience annotations for DAOs. Which of the following are available? Select as many as apply.

- `@Dao`
- `@Insert`
- `@Delete`
- `@Update`
- `@Query`
- `@Select`

Question 2

What are the benefits of using Architecture Components?

- Architecture Components help you structure your app in a way that is robust and testable.
- Architecture Components help you create better UIs.
- Architecture Components provide a simple, flexible, and practical approach to structuring your app.
- If you use the provided libraries and architecture, your app is more maintainable with less boilerplate code.

Submit your app for grading

Guidance for graders

Check that the app has the following features:

- Saves data in at least two different ways.
- Demonstrates how data is preserved differently with different storage options.